BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Lecture Notes

Algorithm Design Using Map and Reduce

This session introduced you to the MapReduce programming model. You learnt how to break a problem to map and reduce components and execute these components parallelly to determine the final solution. The MapReduce programming model is an example of divide and conquer paradigm used for big data processing. MapReduce programming models break a complex problem into smaller tasks called map and reduce. The tasks are executed simultaneously by independent clusters, leading to distributed processing. Because of distributed processing, a significant performance improvement is observed when compared to solving the problem sequentially. This session explained to you the working of map and reduce construct in a step-by-step manner with the help of examples.

Map Construct

We considered a list named LS that had the elements 3, 7, 5, 12, and 8. We would like to square each item present in the last. After squaring, the output list is 9, 49, 25, 144, and 64. A sequential square arrived at by squaring each element, one by one, takes five steps or iterations. In figure 1, f(x) refers to the square function operating on each element of the list LS:

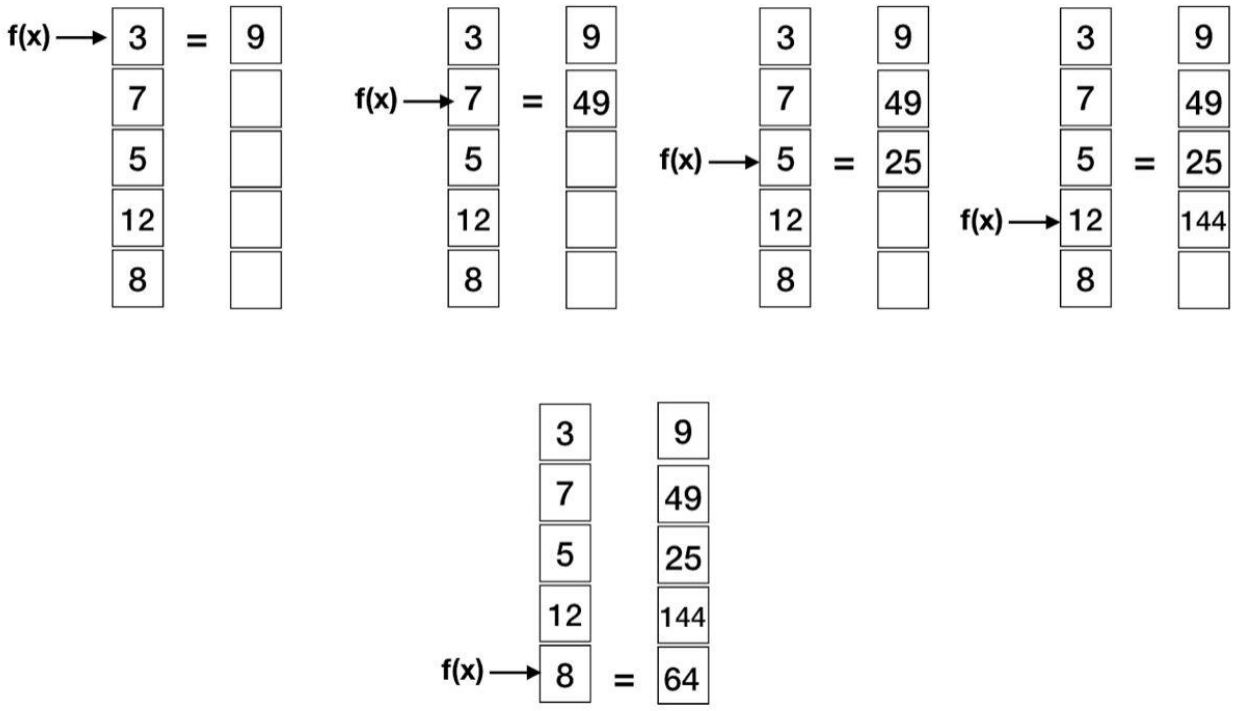


Figure 1: Sequential Squaring

You know that the square function is a unary function. A square operation can be performed independently on each element present in the LS list. The square operation of the i^{th} element does not have any dependency on the output of the square operation performed on the $(i-1)^{th}$ element. Hence, if the square operation is performed on each element of the list at the same time, a valid output for the entire list is produced in one step. Take a look at figure 2 for reference

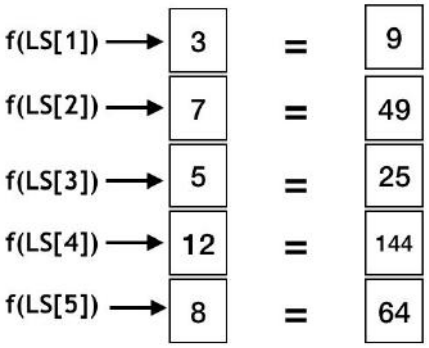


Figure 2: Parallel Squaring

Parallel squaring, thus, can be thought of as a single operation being performed by a parallel distributed model, where a call to a function called MAP(LS) creates 'n' different processes. Each process is assigned an element in the list, and the processes square the number assigned to them. Ideally, a distributed processing model will consist of 'n' processors, where each processor will be assigned a unique process. The 'n' processors execute the processes in parallel. For a better understanding, refer to figure 3.

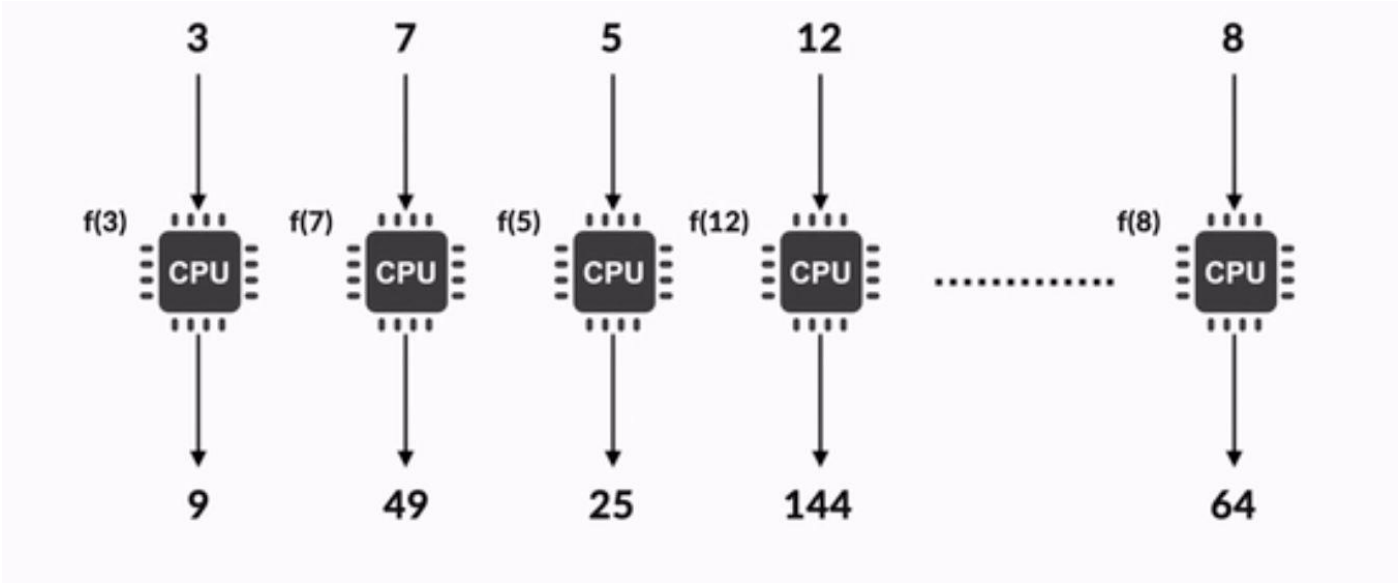


Figure 3: Parallel Distributed Computing Model

As all the ’n’ computations are carried out in one step, the overall time taken by this method is O(1) time. So, the map construct internally breaks a complex task into smaller independent tasks and executes them in parallel using multiple processors, where each processor processes a single task. Ideally, map decides the appropriate number of processes based on the input size, and these processes are assigned to multiple processors to be carried out in parallel.

After understanding the working of a map construct, we discussed how it is used to extract a given keyword from multiple documents in parallel. In this example, the map function extracts the relevant keywords from the given input file. So, if there are numerous documents, the map construct creates

multiple processes equal to the number of documents, which check for the presence of the keyword in all the documents in parallel and extracts them. Refer to figure 4 for a better understanding.

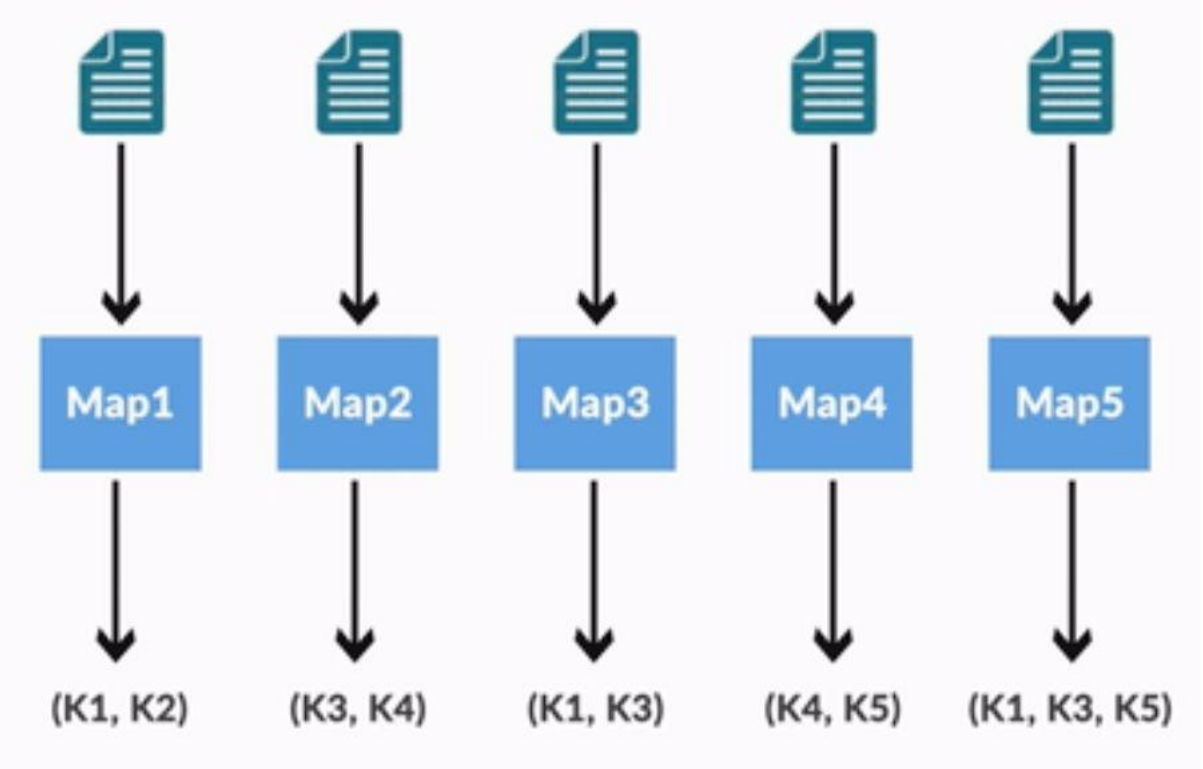


Figure 4: Parallel Keyword Extraction

Therefore, map is assumed as a single line of code or a function call where one function gets repeatedly executed on data distributed across multiple processors. The underlying framework internally creates the required number of processes and assigns these processes to processors for execution.

Performance Evaluation of Map Construct

Let the number of tasks be 'n'.

Then the number of processors is also 'n'.

Let's assume the time taken by each processor to finish a single task to be 't'.

Cost of execution = n x t

The time taken for applying map on 'n' items sequentially = O(n)

Cost of a sequential algorithm = O(n) x 1 processor

Time taken for applying map on 'n' items in parallel = O(1)

Cost of a parallel algorithm having 'n' processors = O(1) x n processors

Reduce Construct

Reduce, as the name suggests, is the act of performing any valid operation in parallel on a chunk of input data and getting an output that is much smaller in size than the input. Reduce construct is a part of the MapReduce paradigm, and both of them together help in solving complex problems.

In the lectures, you learnt how to perform a summation operation on a list of values in parallel. Before that, you saw how numbers in a list are added sequentially. With the help of a sample list having eight numbers, the sequential addition was demonstrated. Refer to figure 5.

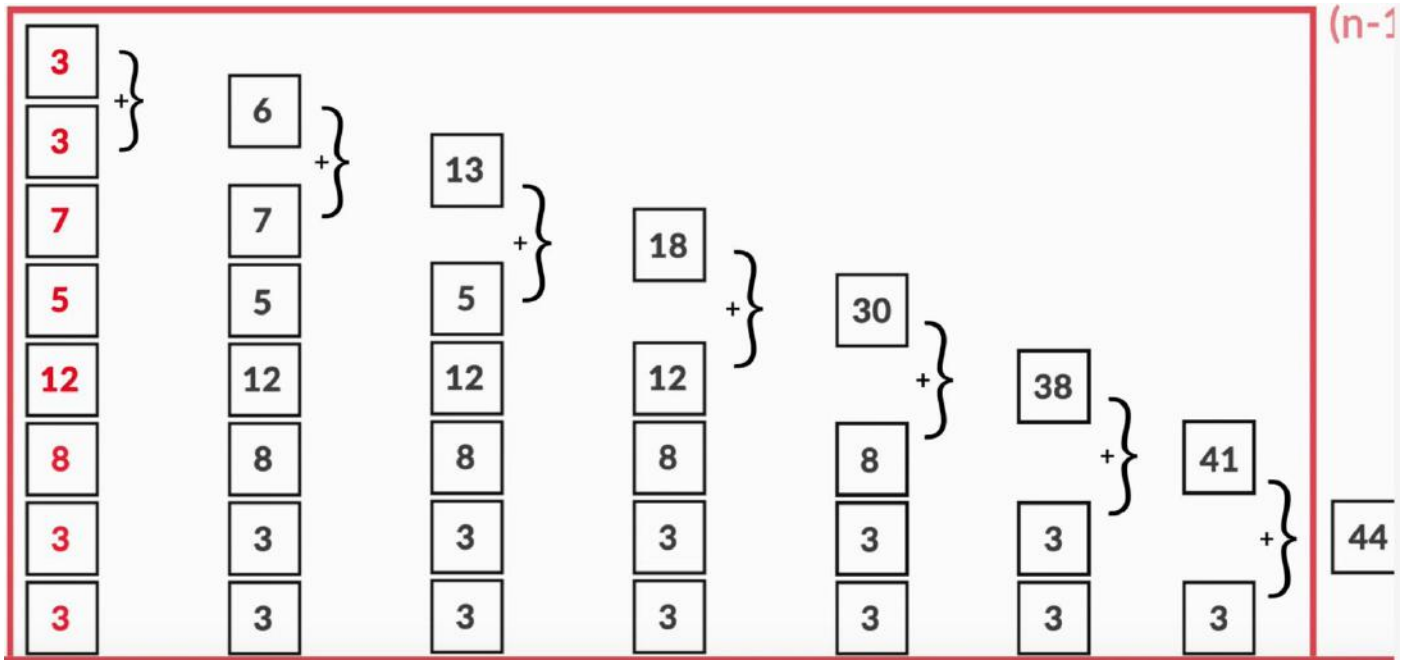


Figure 5: Sequential addition

In the figure above, it is clearly visible that seven steps are required to add six numbers.

To sum all the elements in parallel, we considered the same set of eight elements which was used to demonstrate sequential addition. As addition is a binary operation, in the first step (n/2), four add operations can be performed independently. These four additions will result in four values, which can be added separately using (n/4) two add operations. As in the previous case, these two add operations will generate two values, which can be added using (n/8) one add operation. In sequential addition, seven steps are required to add eight numbers, whereas in parallel addition (log n), three sequential steps are required. Please refer to figure 6.

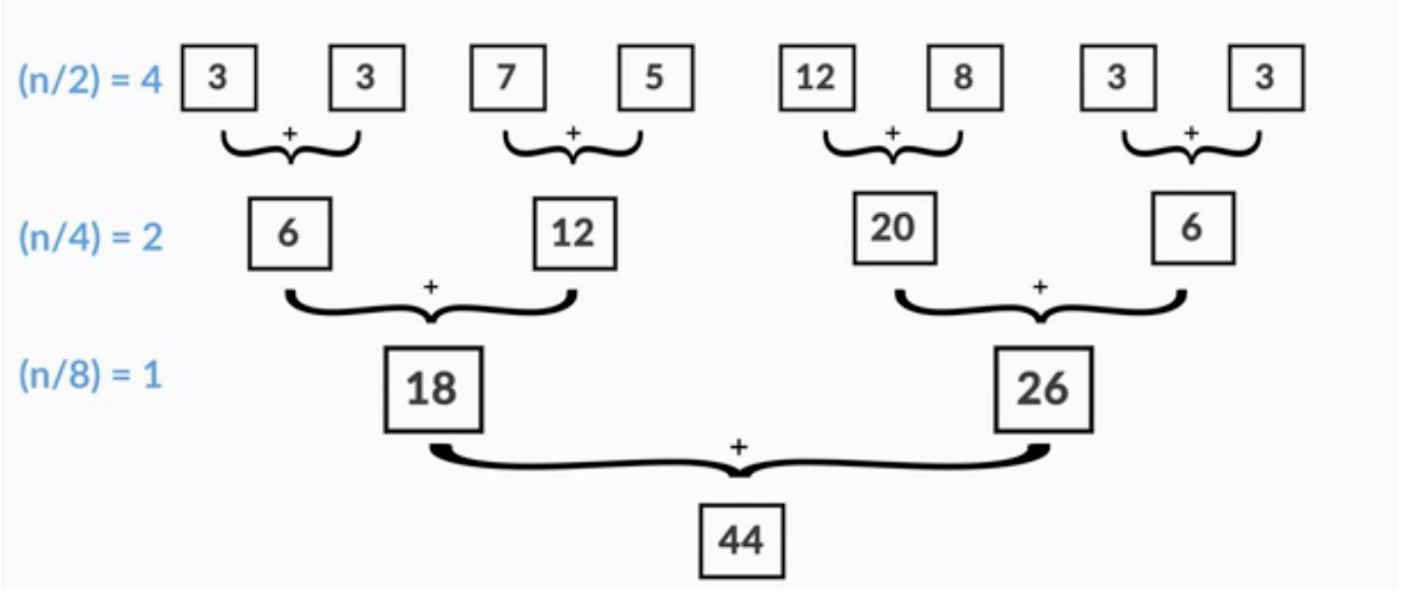


Figure 6: Parallel Addition

In the figure above, each step is comprised of data parallel operations, which means that the operations in each step are performed in parallel. So repeated data parallel operations are performed where the result of the previous step goes as input into the current step, and so on. That is, the output of the (n/2) additions in the first step goes as an input into the (n/4) additions in the next step, and so on. Hence, this sequence of data parallel operations forms a tree in reverse, where the operations are started from the leaf nodes, and the final output is present in the root node. Hence, this process is also known as reverse tree parallel algorithm. Refer to figure 7 for a better understanding.

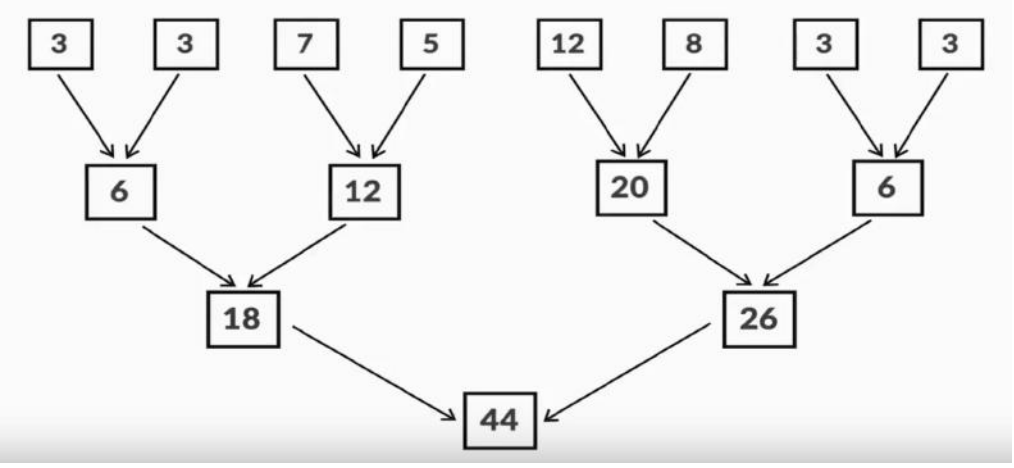


Figure 7: Reverse Tree Parallel Algorithm

Performance Evaluation of Reduce Construct

Let the number of elements to be reduced be 'n'.
For a binary operation, the number of data parallel operations performed in the first step is '(n/2)'.
Hence, the number of processors required is '(n/2)'.

Cost of execution in parallel = O(n/2 x log n)

Cost of execution in sequential algorithm = 1 processor x (n-1) steps

Speedup = (Tseq/Tpar)

Reduce algorithm is more costly because after the first data parallel step, the majority of the processors remain idle in the subsequent steps. But reduce algorithm gives a significant speedup when compared to the sequential method because in the reverse tree algorithm process, eight elements are added in three steps, whereas in the sequential method, seven steps are needed to add eight elements. Hence, in case of reduce construct, a higher cost is incurred to achieve a better performance. Reduce construct is very useful in aggregating results, by performing sum, max, count, etc. operations on a group of data.

Applications of Reduce Construct

In this lecture, you learnt that apart from addition, the following operations can be performed using reverse tree algorithm:

- Finding the max out of a list of 'n' items
- Adding 'n' matrices
- Merge sort

Refer to figure 8 to understand how merge sort uses reverse tree algorithm to sort a list of elements. In each step, multiple pairs of sorted arrays are merged into a single sorted array.

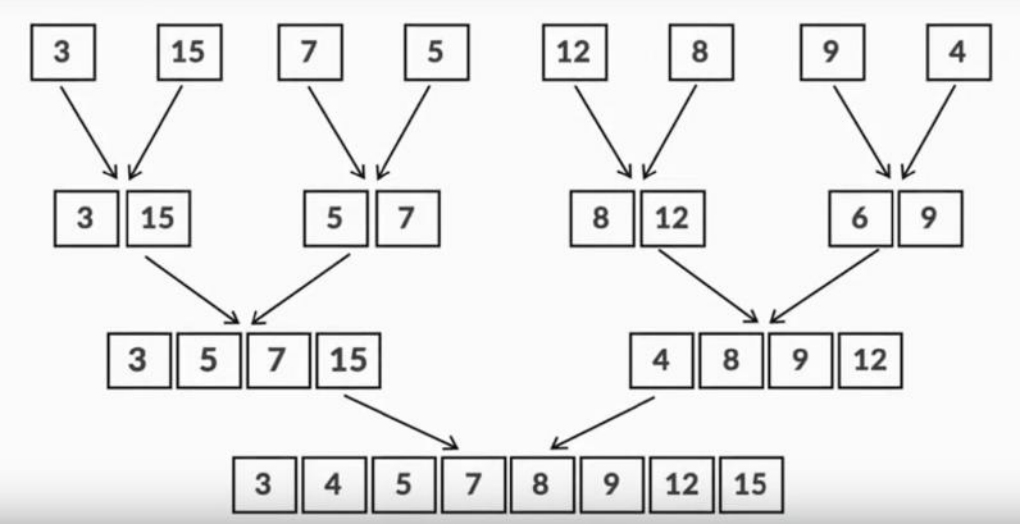


Figure 8: Merge sort

As MapReduce uses distributed processing, independent tasks are spread across a cluster of interconnected machines. This architecture provides a couple of significant benefits. They are —

- **Performance speedup:** As discussed earlier, performance speedup is bound to happen because of parallel distributed computing. If a single computer takes 40 minutes to finish a task, then ten computers working in unison will complete the job in approximately four minutes.
- **Fault tolerance:** A distributed system is fault tolerant because all the worker nodes are directly connected to the master node. Hence, failure of a couple of nodes does not affect the working of remaining active nodes. The cluster is always active and functional even if a couple of worker nodes fail. As MapReduce processing is performed in a distributed cluster, failure of a couple of nodes does not affect the processing. The tasks that were being performed by the failed nodes will be assigned to other active computers in the cluster. Hence, in spite of a couple of node failures, the MapReduce process is completed successfully.

Map Reduce

After understanding the concept of the map and reduce construct, you learnt how map and reduce are used together to solve a problem. In a MapReduce operation, just as in the name, the reduce task always follows the map task. The map task reads the actual input data, processes it, and emits some output. The reduce construct operates on the map output and generates the final output of the entire task.

For a better understanding, we explained MapReduce using the document ranking example. Assuming a collection of documents, with a relevance score computed for each document for a given keyword, we prepared a ranked list in which the relevance scores were used to sort the documents. The relevance scores of each document for a given keyword were computed in parallel during the map phase. The next task was to sort the documents based on the relevance scores. Hence, merge sort was used to sort the documents. As merge sort uses reverse tree parallel algorithm, the merge sort algorithm was implemented in the reduce phase. The final output of the reduce construct was the sorted list of documents based on their relevance scores. Figure 9 is a diagrammatic representation of the document ranking process using MapReduce.

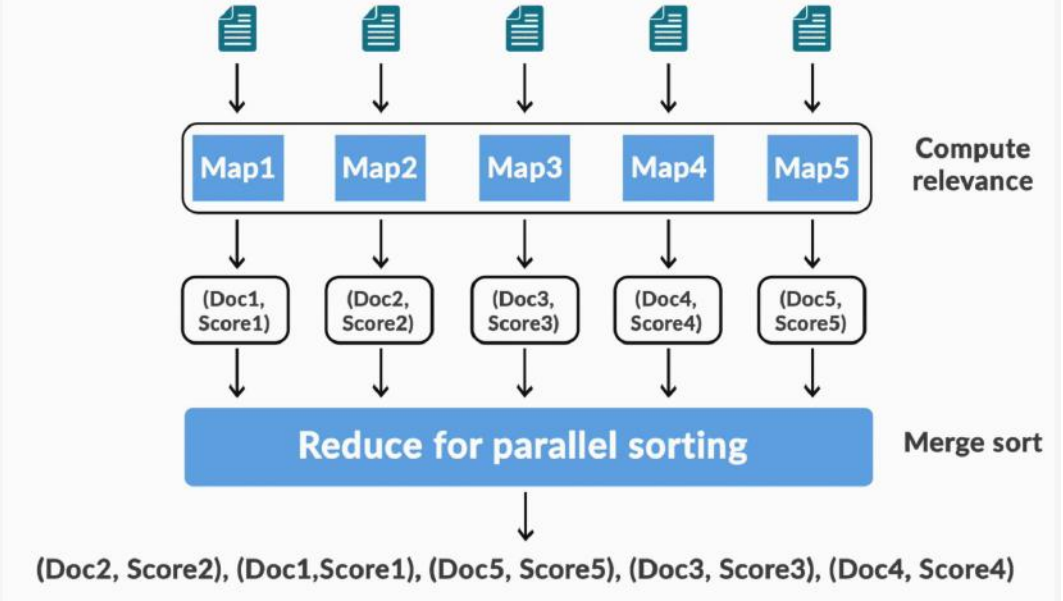


Figure 9: Document Ranking Using MapReduce

Please note that the discussed method is a very old method for ranking documents. Before the advent of Google, the document ranking process was almost the same as what has been discussed in the lecture.

Clustering Using Map Reduce

Clustering is the process of dividing similar data into a fixed number of groups or categories. Similar data records are kept in the same cluster. In this example, the clustering algorithm is considered a black box. In other words, we have a clustering algorithm that takes a dataset as the input and gives 'k' clusters as the output. We are not bothered about how the clusters were formed and the logic used in partitioning the data.

As usual, our entire data was distributed across an interconnected group of processing units. The clustering algorithm was applied in parallel on the data present in each computing machine using the map construct. Let's assume the data is distributed across three machines and the clustering algorithm produces three clusters as output. So, three parallel instances of the clustering algorithm were used to process the documents in parallel. Each instance of the clustering algorithm produced three clusters as output. For example, after applying the clustering algorithm on the first document, clusters C1A, C1B, and C1C were produced as output. Refer to figure 10 for a better understanding.

After the application of clustering algorithm using the map construct, we had three clusters for each document. In our example with three documents, after the application of map construct, we had nine clusters. But our requirement was to divide the entire data set into three clusters. This was achieved by merging all the clusters having similar properties to a single big cluster. In our example, clusters C1A, C2A, and C3A were merged to produce the final cluster named C1. Similarly, clusters C1B, C2B, and C3B were merged to produce the final cluster named C2, while C1C, C2C, and C3C were merged to produce C3. This merge operation was a reduce operation and was applied either sequentially or parallelly to produce the final clusters C1, C2, and C3. Refer to figure 10 for a better understanding.

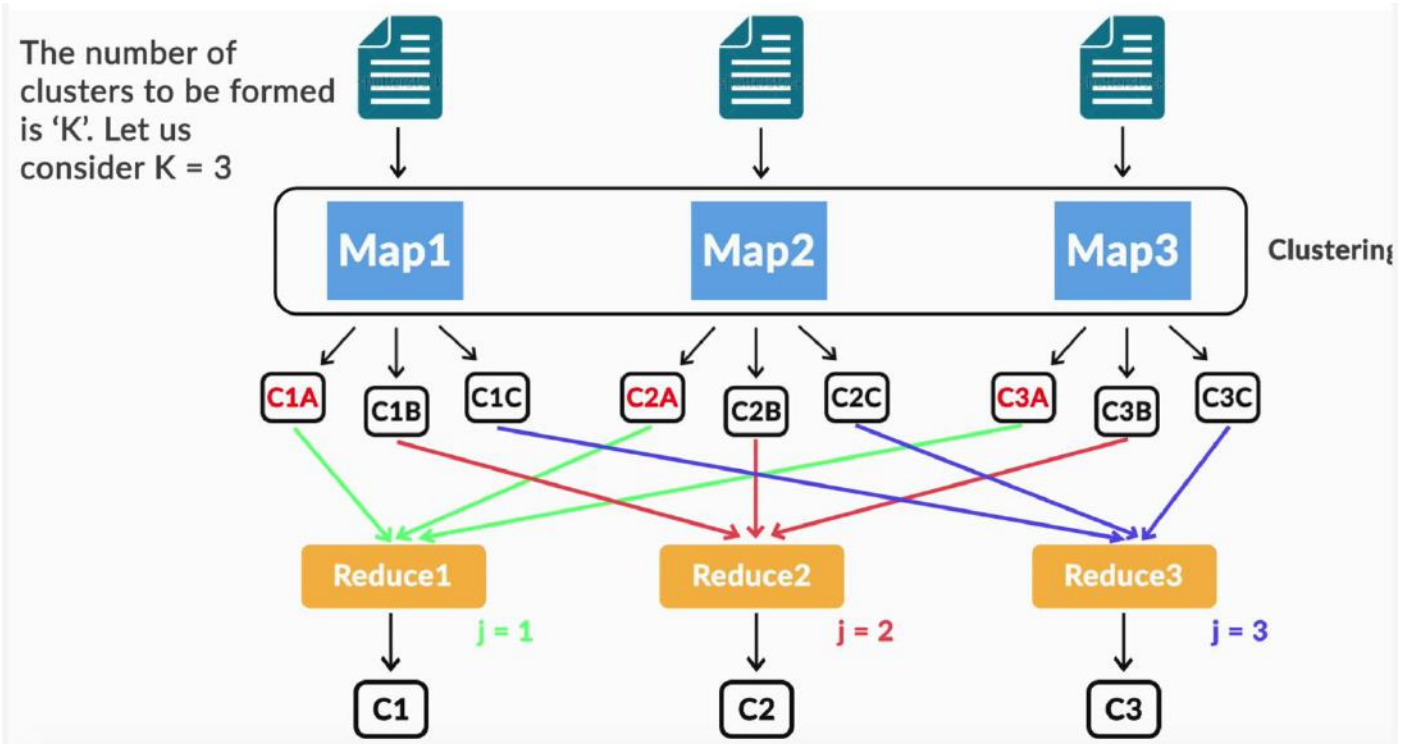


Figure 10: Clustering Using MapReduce

Iterative MapReduce

There exist some problems where applying MapReduce once on a given dataset does not provide the correct result. Hence, you are required to execute the same MapReduce repeatedly till a termination condition is met, or the output converges. A point to be noted here is that the output of the previous MapReduce iteration always goes as an input in the next iteration. This computation paradigm is known as iterative MapReduce.

The concept of an iterative MapReduce was explained using the example of a web crawler. The web crawler works in the following manner. Firstly, a list of URLs is fetched from the database that was reached in the previous search. The documents reachable through these URLs are fetched or downloaded. Secondly, the downloaded documents are scanned, and the crawler retrieves all the URLs present in these documents. Now download the documents corresponding to these newly fetched URLs. Lastly, these steps are repeated till the total number of downloaded documents reaches an upper limit.

So a single iteration in a web crawler can be implemented as a MapReduce algorithm. Initially, a set of documents was downloaded using the URLs fetched from the DB. These documents were distributed across a cluster of processing nodes, and using map construct, these documents were scanned in parallel for fetching all the URLs embedded within these documents. Refer to figure 11 for a better understanding. The likelihood of the presence of duplicate links is high because the map construct simply fetches the link whenever it is encountered while parsing the document, and a single document may have duplicate links. So, to generate a final set of unique URLs, a reduce operation was performed on all the set of URLs produced as output during the map phase. Refer to figure 11 for a better understanding.

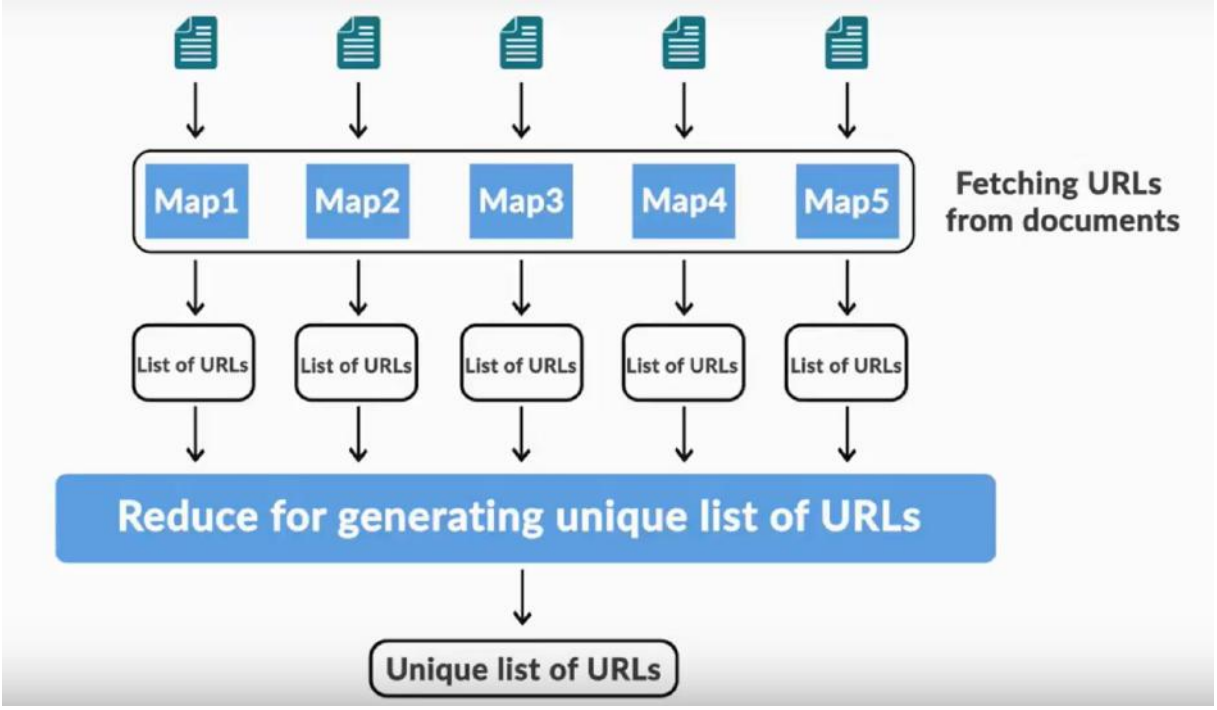


Figure 11: Single Iteration of a Web Crawler Using MapReduce

A flowchart representation of web crawling using iterative MapReduce is mentioned in figure 12 for your reference.

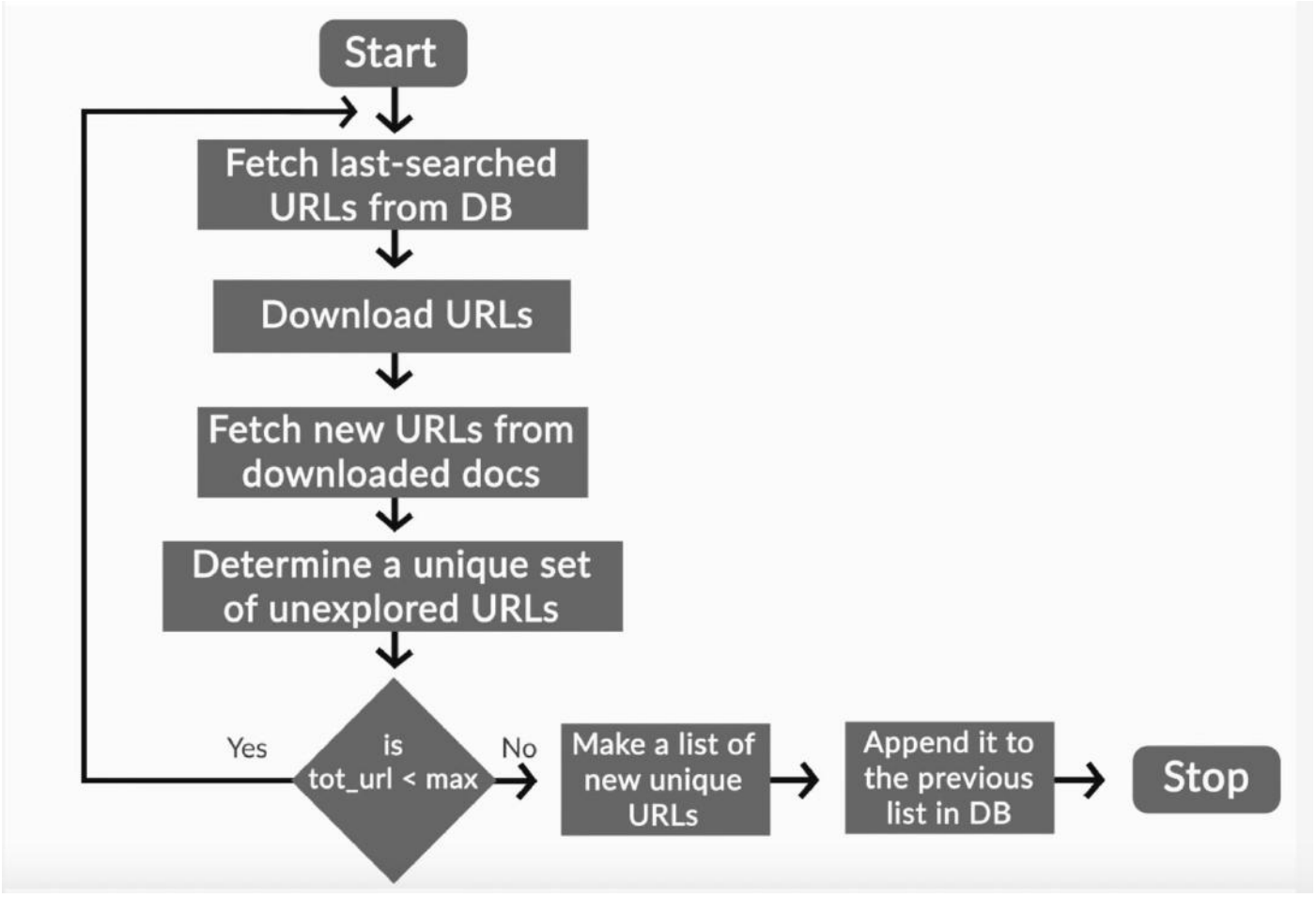


Figure 12: Flow Chart of a Web Crawler Using Iterative MapReduce

UpGrad

Lecture Notes

Data Abstraction

In this module, you learnt about the design principle called Separation of Concerns (SoC) and how the concept of abstraction implements the same.

Separation of Concerns (SoC)

The design principle called Separation of Concerns helps you to separate the concerns of the users from those of the providers (software developer).

Let's take the example of a mobile phone, where the concerns of the user are as follows:

- The phone should produce clear audio
- The phone should not be too heavy
- The phone should not lag
- The phone should be affordable

Whereas the implementer or the provider of the phone is concerned with the following:

- The power consumption
- Where to purchase the various modules of the device from
- Can the cost of components be cheaper?

Now, since the concerns of both the user and the provider are different, the provider needs to take its concerns into consideration and think about their implementations separately. A software designer must identify and separate these concerns, look at the perspective of the user versus the perspective of the provider, and identify what needs to be done for each of these. These concerns form the design elements of any software.

Abstraction

Abstraction is the process of helping the software developer address the concerns of both the user and the provider separately. Using abstraction, you can hide the actual implementation details of the software from the user and only provide him with the ability to use a particular functionality. In simple terms, this means that the user only knows what the software does but not how it does it.

Abstraction example:

You learnt about the auto meter or the fare meter of an auto rickshaw. For a user riding an auto rickshaw, the expectation is that when he presses a button, a ticket with the total fare is printed. This can be implemented as a function named getfare(), which returns the total amount to be paid to the driver. On the other hand, the provider needs to figure out how to compute the fare. The provider needs to consider the base fare, the distance rate (rate

per KM), and the total distance covered by the user and, based on the accumulated information, calculate the total fare. This can be done by writing the formula as shown below:

Total Fare = Base Fare + ((Fare per Km) *Total Distance)

To implement this formula, the provider must calculate the distance, which can be done in multiple ways.
Now, using abstraction, we can hide the details of the actual implementation of these calculations from the user, and just provide him with the total fare.

Data Abstraction

Data abstraction allows you to separate the concerns of users from providers with respect to data.
The user concerns with data are —

- What is the type of data?
- What is the behaviour associated with the data, i.e. what operations can be performed on the data?

The provider's concerns with data are —

- How to represent or store this data?
- How to implement the operations that are associated with this data?

Abstract Data Types (ADT)

An Abstract Data Type (ADT) is a model of a data structure that defines the following:

1. The properties of the collection of data
2. The operations that can be performed on the data

You already know that Abstract Data Types (ADT) help us achieve abstraction of a program. Some other benefits of using ADTs are as follows:

1. It helps understand the code easily
2. It makes it easier to implement changes to the actual data structure code since only a small portion needs to be edited.
3. They can be reused later

A simple way to understand ADTs is to take an example ADT, say, a box with numbers in it. This is similar to the individual word blocks in a Scrabble set. Now, in this box, neither the sequence of numbers matters because both [12, 23, 5, 6] and [23, 12, 5, 6] are the same, and nor does the uniqueness of a number.
Some things (operations) that can be performed with this box are —

1. Add a number to this box: addNum()
2. Search for a number: searchNum()
3. Remove numbers from the box: removeNum()
4. Find the number of items in the box: sizeBox()

Implementing ADTs Efficiently

While implementing an ADT, the provider will have a wide range of data structures with him, and he can implement the ADT using any one of the data structures. You saw how a Stack ADT could be implemented using either an array or a linked list. The important question that needs to be answered is, “How does a provider decide the best possible ADT for a given problem?”

Well, there is no sure shot way to arrive at an answer, but there are some things that the provider should do to arrive at the best possible solution. These are —

- The provider should find out the operations that will be used most frequently for an ADT in any scenario.
- The provider should then consider the cost of implementing the most frequent operations of the ADT using a different data structure and choose the best option.

To better understand this, let's take a look at an example of the dictionary ADT. You can use either an array or a linked list to implement it. Now, you have to finalise one data structure by considering the fact that the most used operations on the dictionary ADT would be 'find' and 'add'. But if you were implementing the ADT for a phone book on a mobile phone, you would be using the 'find' operation far more frequently than the 'add' operation. Hence, it would be better to implement the ADT using a sorted list. Similarly, you can then choose the best possible implementation for different use cases.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Lecture Notes

Data and Distributed Storage

This module introduced you to the storage component of the Hadoop ecosystem, namely, the Hadoop Distributed File System (HDFS) which, unlike an RDBMS, supports the storage of all kinds of data — structured, unstructured, and semi-structured. The first session explained to you the scale at which data is being generated and how to extract valuable information from that data. In the second session, you learnt about the architecture and characteristics of HDFS. It also explained the underlying advantages of HDFS. You also learnt about the rack awareness and optimisation utilities that are provided by the HDFS, such as balancer and disk balancers.

Data and Information

Every day, millions of people across the globe click photographs, create videos, send emails to their friends, send text or voice messages on WhatsApp, browse millions of websites, update their Facebook profiles every one or two hours, and so on. International Data Corporation (IDC) has predicted that by 2020, 1.7 megabytes of data will be generated per second per human across the planet.

Most of the business houses collect data about their consumer preferences, purchase behaviours, etc. to increase their level of engagement with their consumers. It's a great challenge to extract valuable business information from such a huge volume of data. For example, it would be very difficult for an organisation to understand the salary distribution of its employees by analysing their physical pay slips. To find out data such as how many of the employees have drawn a certain amount of their salaries or what is the total amount of income tax paid through the organisation to the government in a particular month towards the earnings of all the employees, the company must use digital data analysis. From this, you should be able to understand that the amount of useful information is several orders of magnitude smaller than the total volume of raw data.

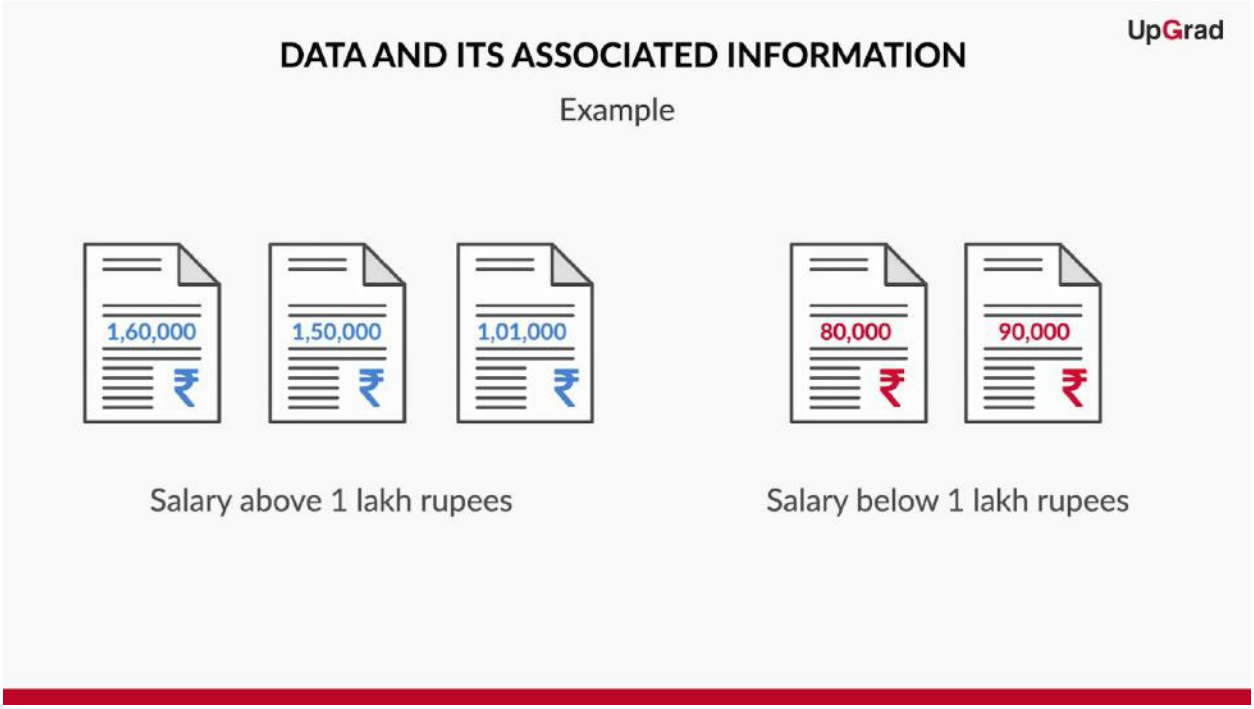


Figure 1: Data and Information

Nature of Data

Data is continuously being generated from several sources, and hence, they differ in nature. Big data is broadly classified into three main categories, namely, structured, unstructured, and semi-structured data.

Structured data refers to any data item that resides in a fixed field within a record or file. Structured data is organised and can be stored in databases, i.e. in tables with rows and columns. Therefore, it presents the user with the advantage of being entered, stored, queried, and analysed easily, using something called the **Structured Query Language (SQL)**. Examples of structured data include Aadhaar data, financial data, the metadata of files, etc. The common applications of structured data, which are in use for several decades, are in banking systems, online reservation systems, inventory control systems, etc.

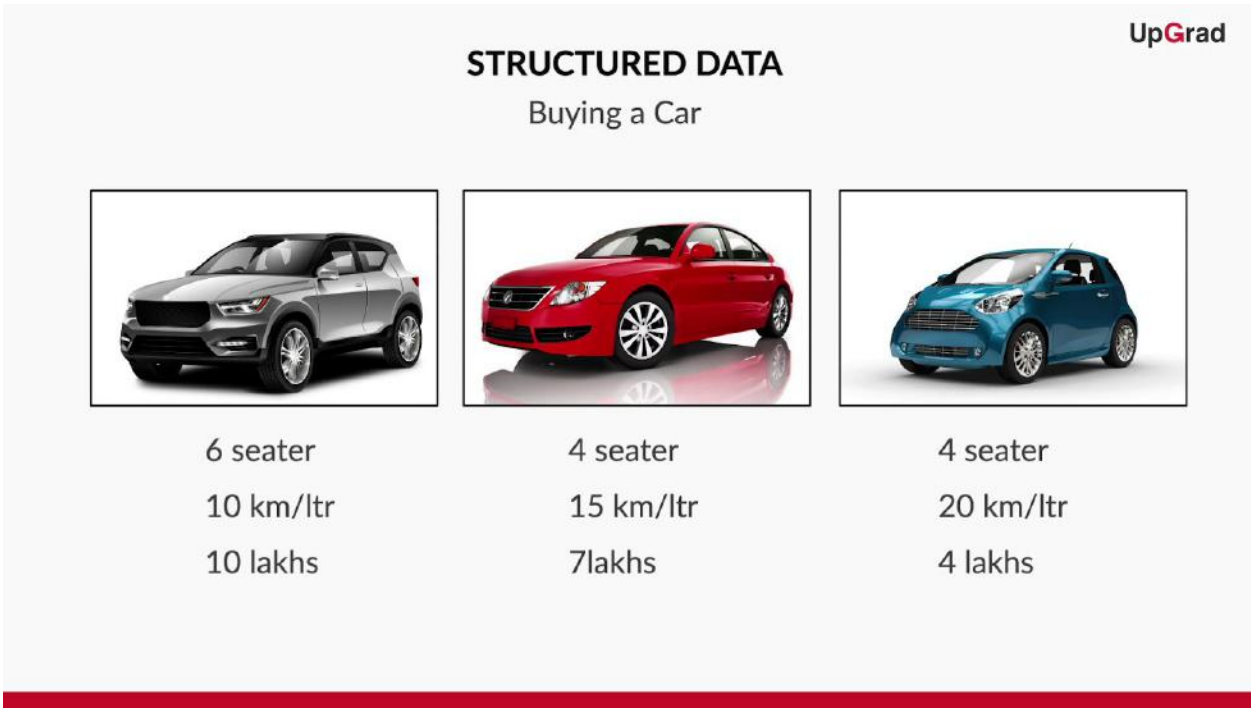


Figure 2: Structured Data

Unstructured data is raw and unorganised. It is not structured via predefined data models or schemas. However, this data may be stored and retrieved using NoSQL databases. Examples of unstructured data include images, audio, video, chat messages, etc., which are usually generated and consumed by humans. It's not surprising that a majority **(~80%) of the data** being generated in today's world is unstructured in nature.

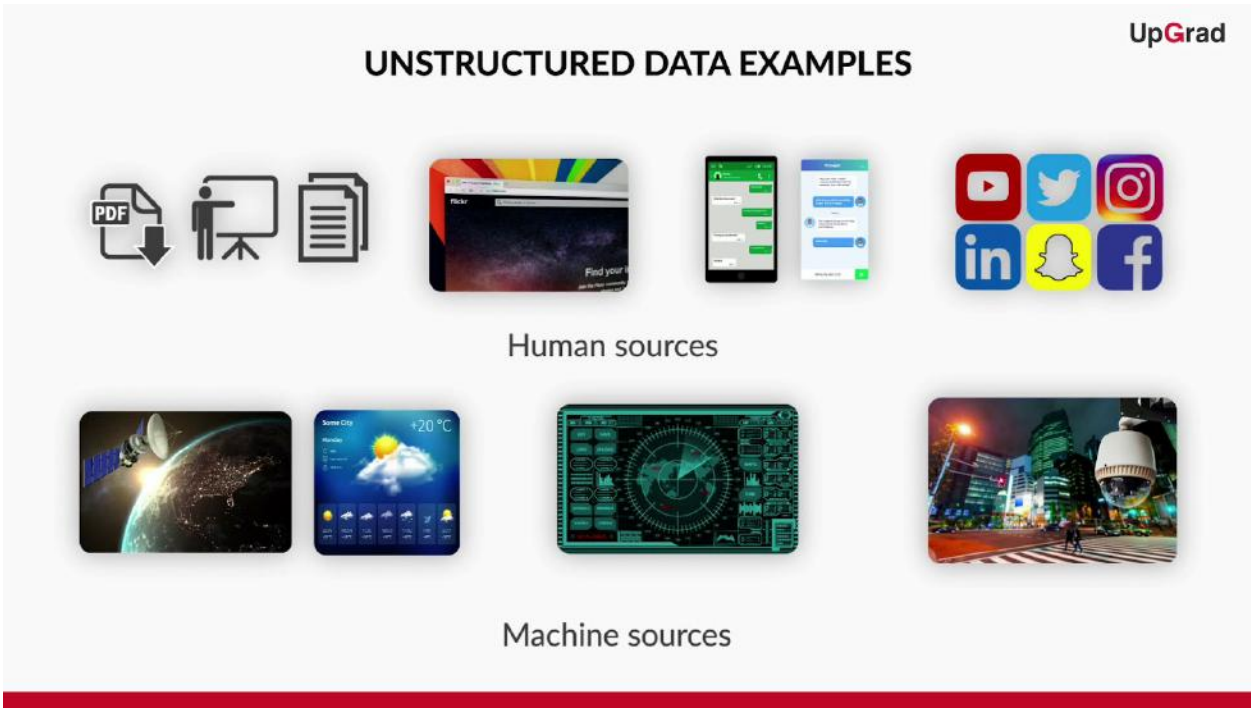


Figure 3: Unstructured Data

Semi-structured data maintains internal tags and markings that identify separate data elements; this enables information grouping and the creation of hierarchies among the elements. There's **no predefined schema** for semi-structured data. In terms of readability, it sits between structured and unstructured data. XML and JSON files are examples of semi-structured data.

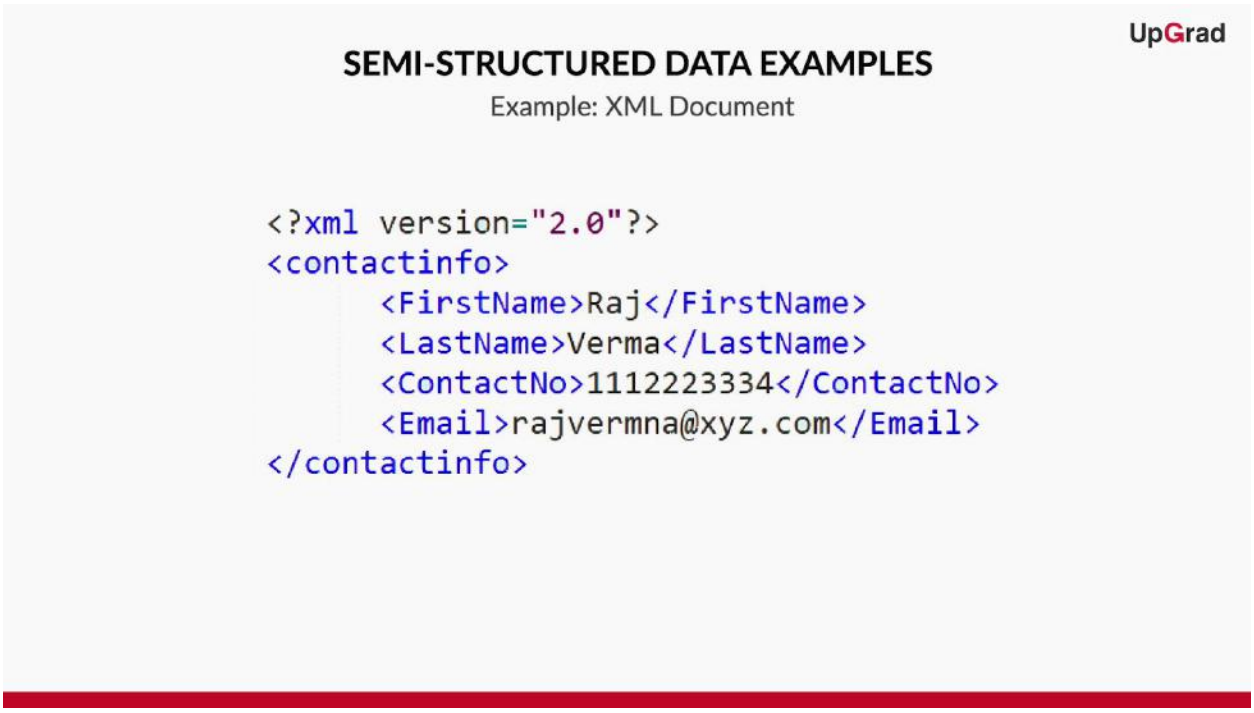


Figure 4: Semi-structured Data

Use Cases of Big Data

The retrieval of information from big data analyses (with structured, unstructured, and semi-structured data) has many applications across fields.

Cognitive IOT System:

Consider the scenario of a campus trying to build an intelligent cognitive IoT system that helps to minimise the power consumption. The information retrieved from the data collected from a host of sensors can assist in making decisions to reduce power consumption.

For a system that helps in making decisions about whether to turn on all the street lights or only a few of them in the campus, the data is collected from different sources, as shown in the image given below. The various data sources include light sensors, motion sensors, students timetables, and other external events (e.g. transport strike in the city).

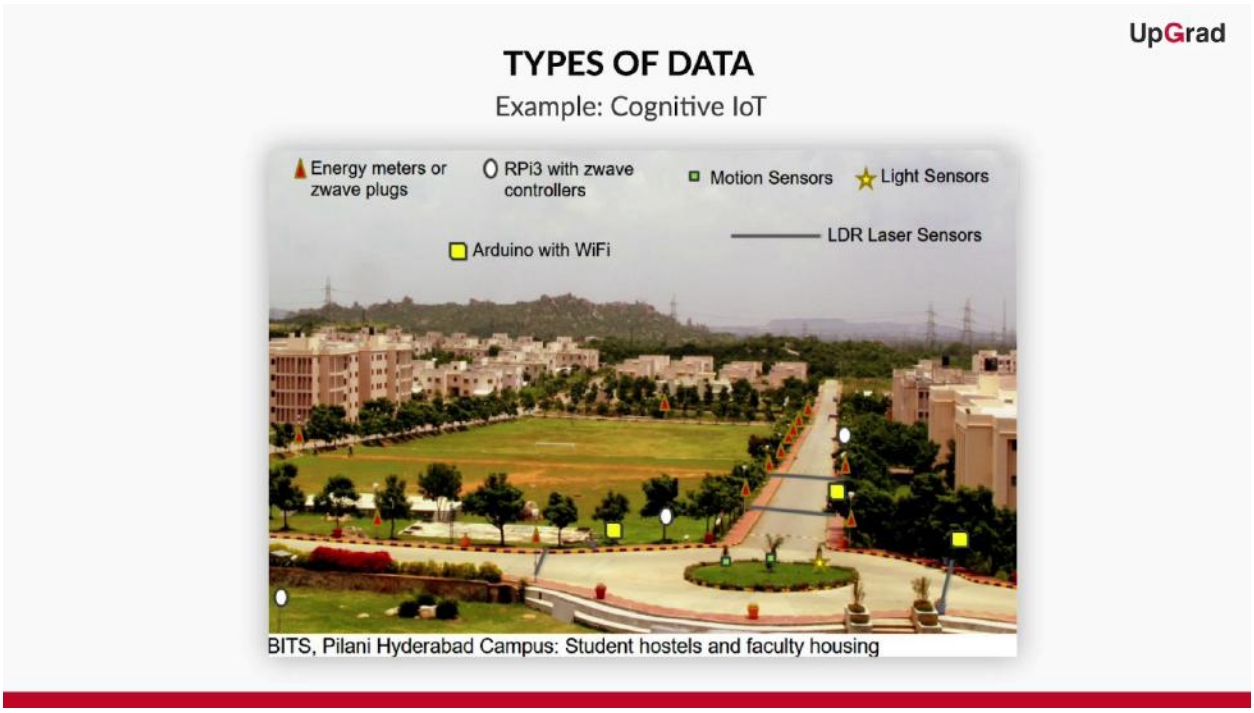


Figure 5: Cognitive IOT System

The big data framework can be used to identify complex events (in this case, the decision to either turn on all the street lights or a few of them) from simple events (like collecting data from different sources) that are triggered by sensory inputs, giving cognitive capabilities to the IOT nodes.

Sentiment Analysis:

Many e-commerce organisations today adopt sentiment analysis to improve their business, brand, and reputation. Objective feedback helps them in identifying the flaws in a product or service, and thus, gives them a chance to improve. It is vital for organisations to understand and analyse what their customers say about their products or services to ensure customer satisfaction.

Sentiment analysis based on **big data information retrieval** from several resources helps in enhancing the reputation of an organisation. It helps to improve customer experience and monitor the online reviews posted by customers on various channels, thus aiding the company in identifying the issues faced by a customer and solving them as early as possible.

Sentiment analysis also helps to evaluate the opinion on a product or a brand and tells whether a particular brand or product is being discussed as well as what is being said about it.

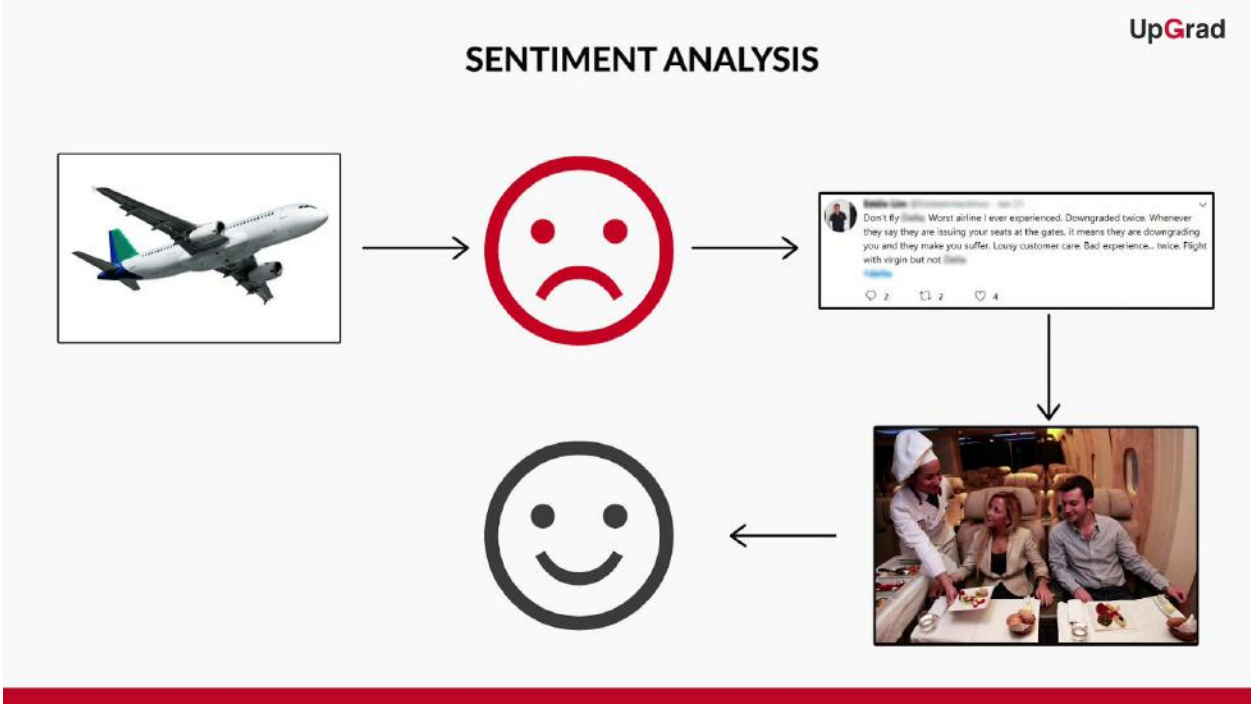


Figure 6: Sentiment Analysis

Characteristics of Distributed File Systems

Distributed file systems have the following characteristics:

- **High availability:** Distributed file systems allow users to store data across multiple nodes or machines in a cluster, thereby providing high availability
- **Concurrency:** They allow multiple users to access data simultaneously
- **Client/server architecture:** One or more central servers store files that can be accessed by remote clients of the network with proper authorisations.

- **Transparency:** It can organise and display data in such a manner that users feel as if the data is stored on a single machine.

Examples: Sun Microsystems' Network File System (NFS) is a traditional DFS, i.e. it doesn't provide features such as fault tolerance. Whereas Google File System (GFS) and **Hadoop Distributed File System (HDFS)** are cluster-based DFSs that were designed to replicate data and provide fault tolerance.

HDFS and Its Advantages

Hadoop distributed file system or HDFS is a Java-based DFS that allows you to store large volumes of data across multiple nodes in a Hadoop cluster. HDFS is very effective for big data storage due to the following reasons:

- **Handles huge volumes of data:** It can handle terabytes and petabytes of data with ease as compared to traditional RDBMSs, which are capable of handling only a few gigabytes.

- **Supports storage of unstructured data:** HDFS support the storage of unstructured data as opposed to traditional RDBMSs, which only stores structured data.

- **Distributed computation:** Since data is stored across several machines, it allows us to take advantage of distributed computation. In this computation, each node can parallelly process the data residing on it.

- **Horizontal scaling:** This allows us to add more machines to a cluster or system and improve the overall performance without having to shut it down, as opposed to vertical scaling.

Architecture of HDFS

The HDFS has a Master-Slave architecture, where the 'Master' is the NameNode, and the 'Slave' is the DataNode. It consists of the following components:

NameNode:
The HDFS architecture is designed in such a way that the user's data never resides on the NameNode. But it records the metadata of all the files stored in the cluster, which includes

location data and the details of the data blocks stored, e.g. size of the file, permissions, hierarchy, replicas, etc. This metadata in the NameNode helps in faster retrieval of the data. The two files associated with the metadata are —

- **FsImage:** This file contains the complete information on file-to-block mapping, block-to-DataNode mapping, directory hierarchy, and so on. It basically contains the complete state of the system at any point in time.

- **EditLogs:** This file contains all the changes/updates to the file system with respect to the latest FsImage. When the NameNode starts, the existing FsImage and EditLogs on the disk are combined into a new FsImage file and loaded into memory. Next, fresh EditLogs are initiated.

A NameNode should be deployed on reliable hardware as it is the centerpiece of an HDFS. It manages the slave nodes and assigns tasks to them. The different tasks of NameNodes include —

- Regulating client access to files
- Managing the file system namespace
- Executing the file system operations, such as naming, closing, and opening files and directories

A NameNode also takes care of the replication factor of all the blocks. The NameNode regularly receives heartbeat and block reports for all the data nodes and ensures that they are alive and acts in case of over-replication or under-replication of data blocks.

DataNodes:
DataNodes have the following features:

- They store the complete data on the HDFS
- They send heartbeat and block reports to the NameNode to indicate data node states
- They create, replicate, and delete data blocks according to the instructions provided by the NameNode
- They are provided with a large amount of hard disk space because DataNodes actually store the data. Since their reliability is not a major concern, commodity hardware is generally used.

In HDFS, when a NameNode starts, it first reads the HDFS state from the ‘FsImage’ and applies the edits to it from the ‘EditLogs’ file, and then it writes a new HDFS state to the FsImage. After

that, it starts the normal operation with an empty 'EditLogs' file. Sometimes, the edit file may be too large, resulting in a side effect: the next restart of NameNode would take longer. The 'EditLogs' often have a large amount of data and combining them with the 'FsImage' would take a lot of time. The secondary NameNode helps in solving this issue.

Secondary NameNode:
The secondary NameNode is just a helper node to the main node. It regularly gets EditLogs from the NameNode, applies them to the FsImage, and then moves the updated FsImage back to the NameNode. The secondary NameNode is also known as the Checkpoint node.

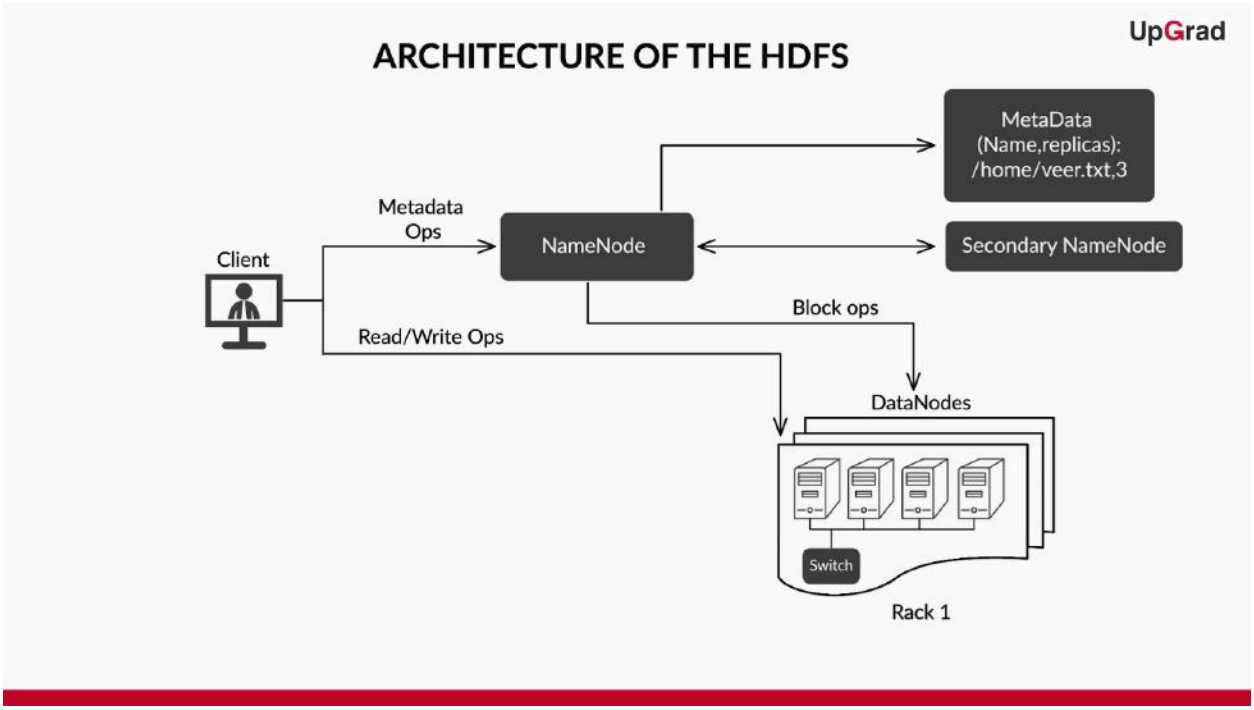


Figure 7: Architecture of the HDFS

Note: The secondary NameNode cannot replace the NameNode, and hence, Hadoop version 1 is considered a single point of failure (SPOF) system.

Standby NameNode:
Hadoop version 2 provides a high-availability feature with a standby NameNode that stays in synchronisation with the main NameNode. It is a backup for the main NameNode and takes over the duties of the main NameNode in case it fails. In fact, a secondary NameNode is not required if a standby NameNode is used.

The two main points to be followed for maintaining consistency in the HDFS are —

1. Active and standby NameNodes should always be in sync with each other, which means that they should have the same metadata. This provides fast failover.
2. And secondly, there should be only one active NameNode to avoid the corruption of the data.

HDFS allows replication of data blocks stored in data nodes to avoid data loss if the data node fails. When the data is being divided into blocks for storage spaces on the HDFS, instead of storing a block of data on just one node, each block is replicated a specific number of times. This way, more than one block replica is stored on the HDFS across multiple nodes.

You can specify the **replication factor** or the number of replicas that need to be created. The default replication factor is 3, which is configurable. The NameNode receives block report from the DataNode periodically to maintain the replication factor. When a block is over-replicated/under-replicated, the NameNode adds or deletes replicas as needed. Increasing or decreasing the replication factor has an impact on the performance of the Hadoop cluster. Over-replication can lead to data redundancy as the NameNode needs to store more metadata about the replicated copies. So, although increasing the replication factor improves data reliability, it has a positive impact only when enough resources are available for storage and processing.

Data Storage in HDFS

The HDFS splits huge files into smaller chunks, which are known as blocks. In general, in any file system, data is stored as a collection of blocks. Similarly, the HDFS splits huge files and stores them as blocks. The default size of a block in Hadoop 2.x is 128MB, and it can be configured as per the requirement. In the collection of blocks, all the blocks except the last block will be of the same size, and the last block can be of the same size or smaller in size.

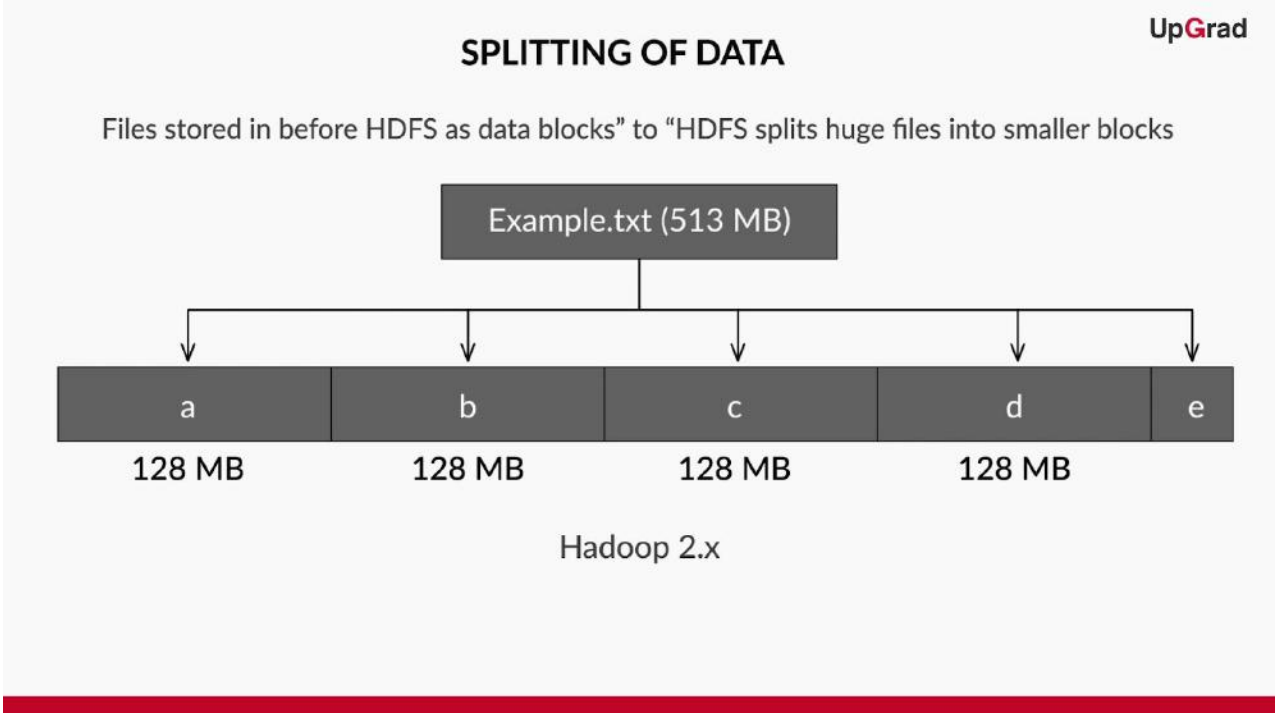


Figure 8: Splitting of data in HDFS

The default size of blocks is large in HDFS because it handles huge datasets. If the block size is small, managing the number of blocks and metadata will create a huge overhead. So, in order to avoid that, the block size is designed to be large. Both increasing and decreasing the block size have their own merits and demerits. The merit of increasing the block size is that it reduces the size of the metadata. On the other hand, its demerit is that it reduces the number of storage nodes, which, in turn, can reduce the throughput for parallel access. Decreasing the block size increases the seek overhead, which, in turn, increases the number of tasks. It also increases the size of the metadata stored by the NameNode.

Read and Write Operations in HDFS

Read Operation in the HDFS:

To read a file from the HDFS, a client needs to interact with the NameNode, as it stores all the metadata. NameNodes check for the required privileges. It also provides the address of the data nodes where a file is stored. After that, the client will interact directly with the respective data nodes and read the data blocks. The read operation in the HDFS is distributed, and the client reads the data parallelly from the DataNodes.

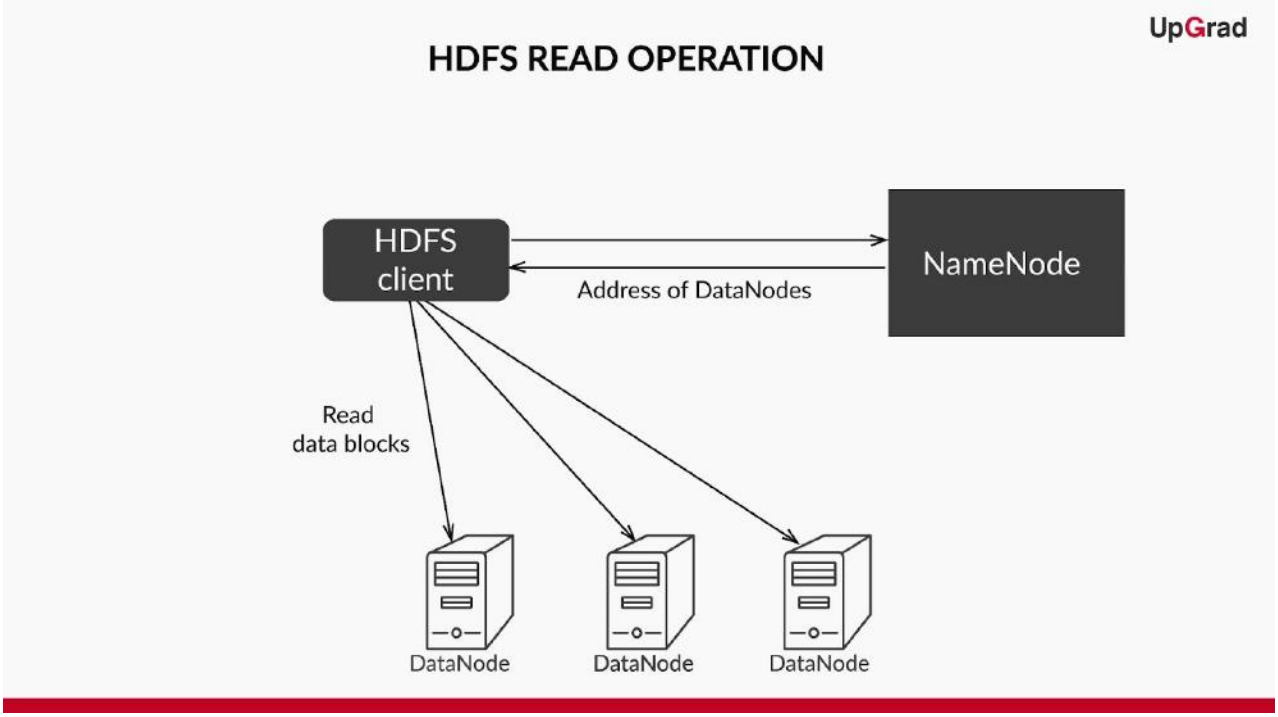


Figure 9: HDFS read operation

Workflow of the Read Operation in HDFS:

1. The client can open any file by calling open() on the file system object. For the HDFS, this is an instance of the distributed file system object.

2. The distributed file system object calls the NameNode using a remote procedure call, or RPC, to determine the locations of the blocks in the file. For each block, the NameNode returns the addresses of the DataNodes that have a copy of that block, after sorting the DataNodes by their proximity to the client.

3. The distributed file system object returns an FSDatainputstream object to the client for reading data. FSDatainputstream wraps the DFSinputstream, which manages the DataNode and NameNode I/O. The client calls read() on the stream. The DFSIputStream, which had previously sorted the data node, connects to the closest DataNode.

4. Data is streamed from the DataNode to the client by calling the read() method repeatedly on the stream. When the block ends, the DFSInputStream will close the connection to the DataNode and then finds the best data node for the next block.

5. 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 the failed DataNodes so that it doesn't revisit them for later blocks.

6. The DFSInputStream also verifies the checksums to check whether the data is corrupt or not. If a corrupt block is encountered, the DFSInputStream is responsible for reporting the same to the NameNode. The DFSInputStream notifies the NameNode before it tries to read a replica of the block from a different DataNode.

7. Once the client has read all the data, it calls the close() method on the stream.

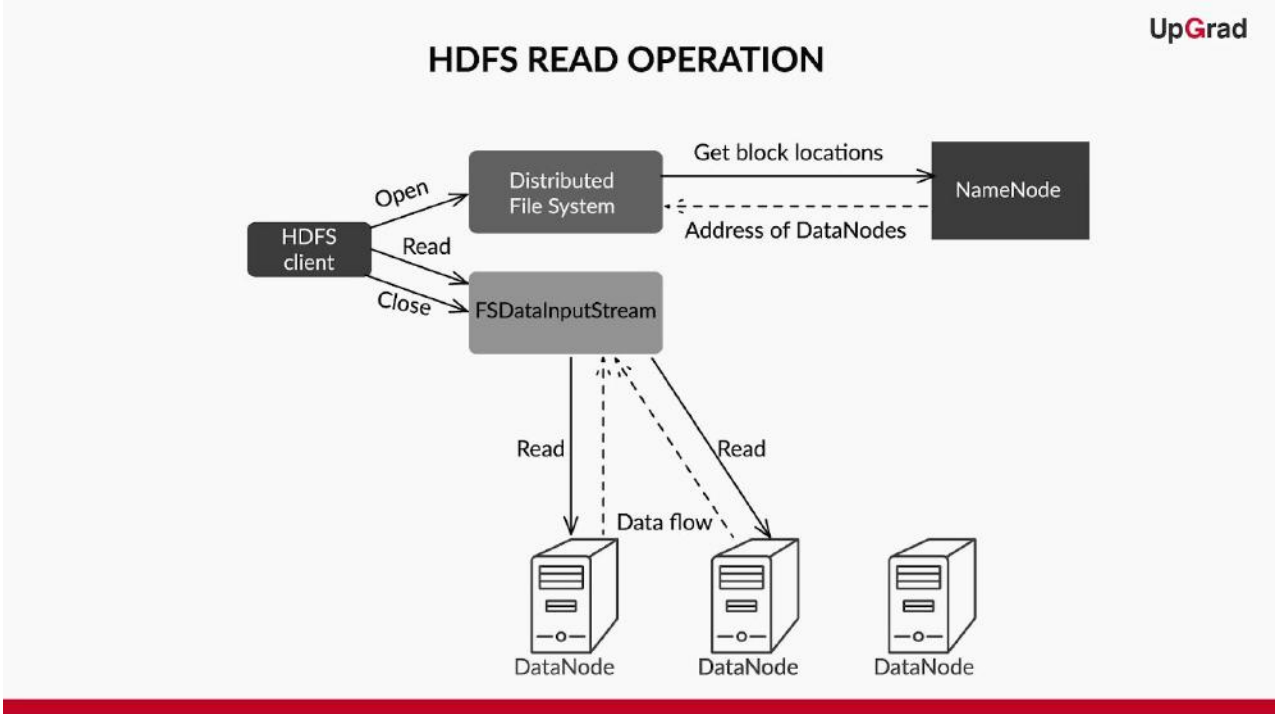


Figure 10: HDFS read operation

Workflow of the Write Operation in the HDFS:

1. The HDFS client initiates a write operation by calling the create() method of the distributed files system object.

2. To create a new file, the distributed file system object makes an RPC call to the NameNode. When the checksum validations are successful, then only the NameNode

writes a record corresponding to the new file. If the checksum validation fails, the file creation will also fail, and the client will be thrown an IOException.

3. The distributed file system object returns an FSDataOutputStream object for the client to start writing data to the HDFS.

4. While the client writes the data, the DFSOutputStream splits it into packets, which are written to an internal data queue.

5. This data queue is then used by the datastreamer, which asks the NameNode to allocate new blocks. This is done by picking a set of suitable DataNodes to store replicas.

6. The DFSOutputStream also maintains an internal queue of packets called the 'ack' queue. These packets from the queue are removed only when they are acknowledged by the DataNodes in the pipeline. DataNodes send the acknowledgement once the required replicas are created.

7. Similarly, all the blocks are stored and replicated on different data nodes, and the data blocks are copied in parallel.

8. Close() is called on the stream when the client has finished writing the data. This leads to flushing of all the remaining packets present in the DataNode pipeline and waiting for acknowledgements before contacting the NameNode to signal that the write operation is complete.

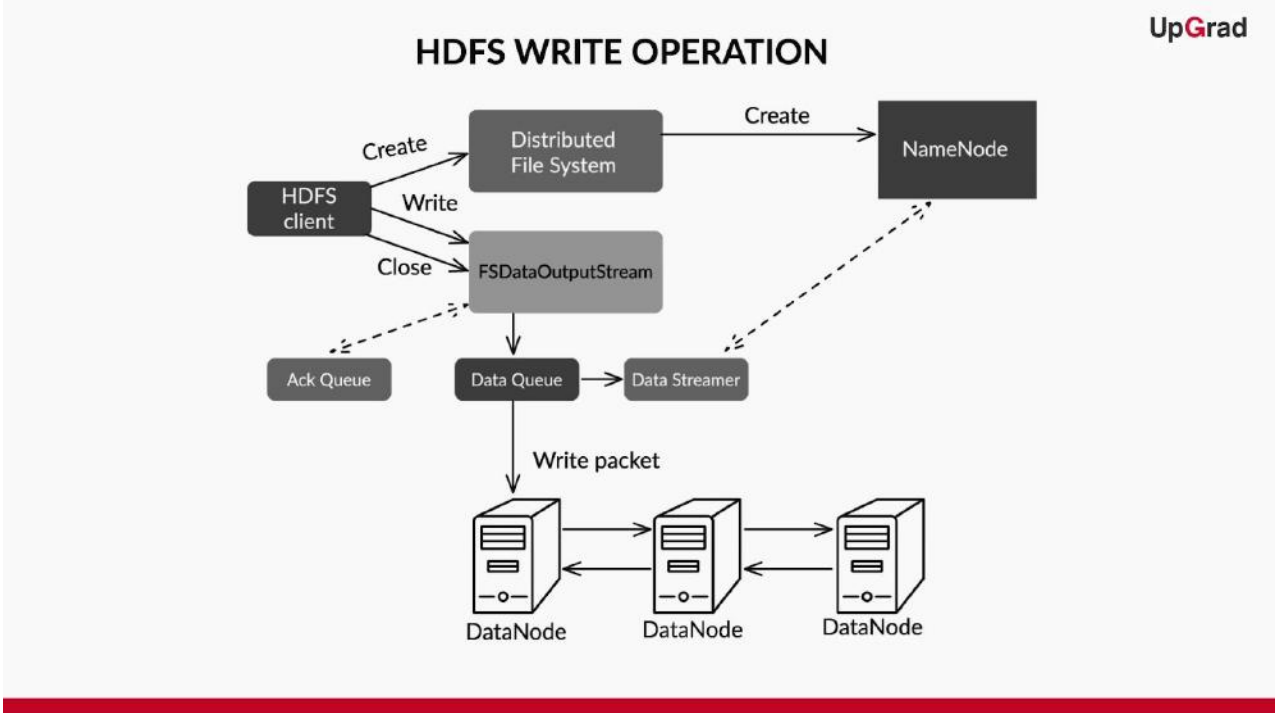


Figure 11: HDFS write operation

Note:

- If a DataNode fails while data is being written to it, the pipeline is closed, and any packet remaining in the ack queue is added to the front of the data queue. Post this, the current block of the good DataNode is given a new identity, which is communicated to the NameNode so that the partial block of the failed DataNode is deleted during the failed DataNode recovery later on. The DataNode that fails is removed from the pipeline, and the remainder of the data is written to two working DataNodes in the pipeline.

- If the NameNode notices that the block is under-replicated, then it arranges additional replicas to be created on another node. Then, it treats the subsequent blocks as normal.

- It is highly unlikely that multiple data nodes will fail while the client writes to a block. As long as it writes a minimum number of replicas, i.e. 1, the write operation will be successful, and the block will asynchronously replicate across the cluster until it achieves the target replication factor.

Features of HDFS

Some important features of the HDFS are —

- **Cost-effective:** Since the HDFS is generally employed on everyday commodity hardware, it is very economical in terms of cost and can be scaled horizontally without investing a large amount of money.

- **Variety and volume:** The HDFS allows you to store different types of data — structured, semi-structured, and unstructured — besides allowing you to store huge amounts of data as well.

- **Data locality:** In traditional systems, data was first moved to the application layer and then processed. However, due to the huge amount of data generated today, transferring it to the application layer causes high latency. To avoid this, Hadoop does the computation on the data nodes by bringing the processing units to these nodes.

- **Reliability and fault tolerance:** HDFS constantly checks the integrity of data stores. If it finds any fault, it reports the same to the NameNode, which creates additional replicas and deletes corrupted files.

- **High throughput:** Hadoop provides high throughput by processing data parallelly.

Note: Though commodity hardware makes HDFS horizontally scalable and fault-tolerant, you might require slightly different and better systems than commodity hardware to solve problems that use techniques other than batch processing.

Concepts That Enhance the Performance of HDFS

A collection of DataNodes is called a rack, and the nodes on a rack are physically stored in close proximity, all of them connected to the same network switch.

Rack Awareness:
It is a concept that helps prevent the loss of data even if an entire rack fails. The policy of rack awareness is, “No more than one replica is stored in one node, and no more than two replicas are placed on the same rack.”

The NameNode ensures that all the replicas are not stored on the same rack. It follows the underlying, built-in rack awareness algorithm to provide fault tolerance and reduces the latency. If the replication factor is 3, the rack awareness algorithm states that the first replica of a block will be stored on the same local rack, and the next two replicas will be stored on a different remote rack. Note that both the replicas will be stored in different DataNodes within that remote rack.

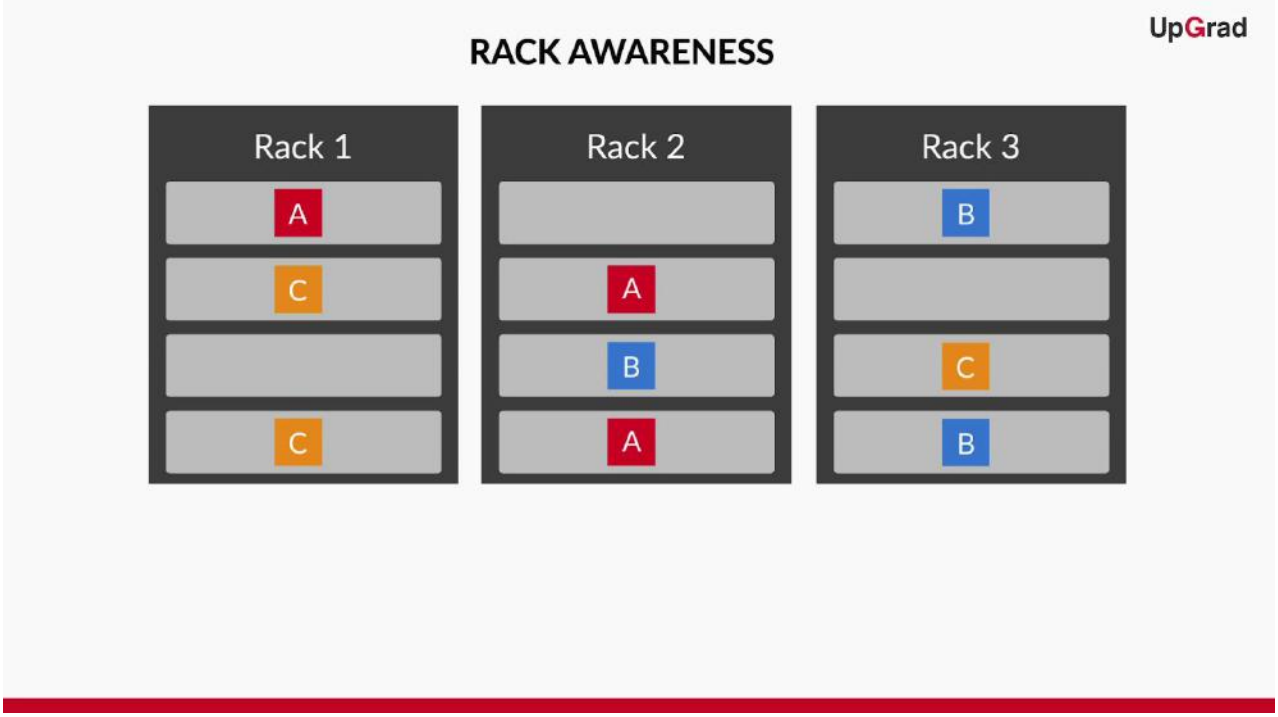


Figure 12: Rack awareness

The advantages of the rack awareness concept are that it improves the network performance by ensuring better network bandwidth between the machines on the same rack than the machines residing on different racks. This has a constraint that the number of racks used for block replication should be less than the total number of block replicas.

Load Balancing:

When a new rack full of servers and networks is added to a Hadoop cluster, data doesn't start spreading to it automatically. The added servers remain idle with no data. The servers on the new rack are then assigned tasks, but with no data residing on them. Hence, they need to grab data from the network, which results in higher network traffic and a slower task completion rate.

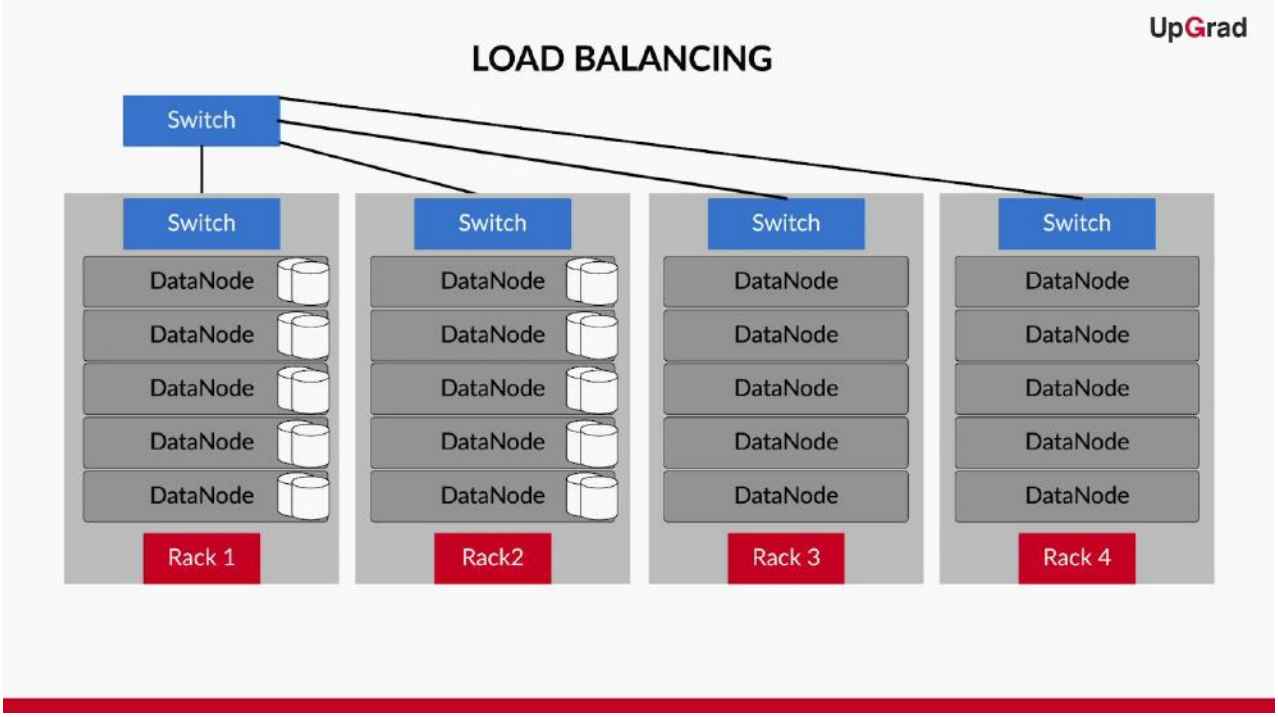


Figure 13: Load balancing

Hadoop has an **HDFS balancer**, which balances the data among data nodes and racks. This balancer solves the aforementioned problem of load balancing. The system administrator, whenever required, can run the balancer using the command line. The command to run the balancer is '**sudo -u hdfs hdfs balancer'**. This command runs the balancer with a default threshold of 10%, which means that this command will ensure that the disk usage for each DataNode differs from the overall usage in the cluster by no more than 10%.

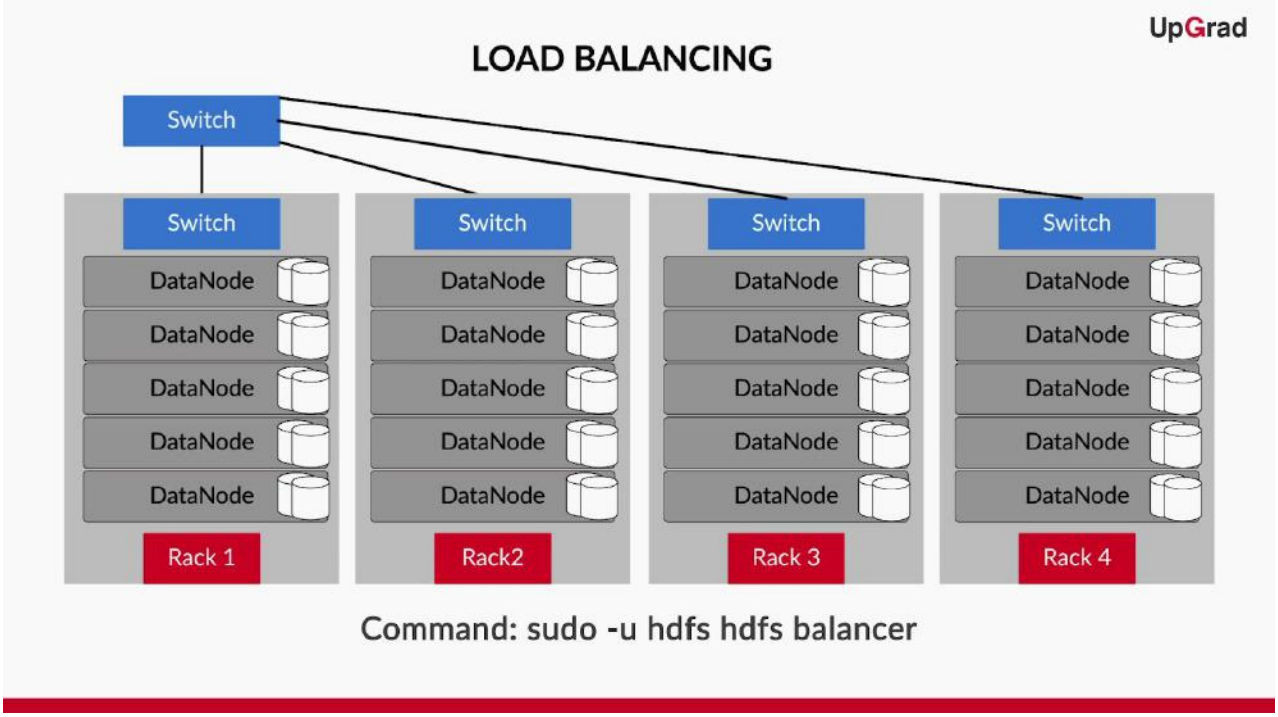


Figure 14: Load balancing

One can set the threshold by running the command '**sudo -u hdfs hdfs balancer -threshold X**'. This command tells the HDFS that each data node's disk usage must be within X% of the clusters overall usage. The command to adjust the network bandwidth before you can run the balancer is '**dfsadmin -setBalancerBandwidth newbandwidth**' (in case of low network traffic of the balancer).

Note: The balancer will take a lot of time when it's run for the first time.

HDFS Disk Balancer:

Hadoop has another balancer called the 'HDFS disk balancer' to solve the problem of non-uniform distribution of data on the disks of a particular DataNode. The HDFS may not always place data in a uniform way across the disks due to a lot of write and delete operations or disk replacement reasons. This leads to a significant imbalance within the DataNode, and this situation is handled by the new intra-DataNode balancing functionality called the HDFS disk balancer. The disk balancer works on a given DataNode and moves blocks from one disk to another.

When a new block is written to the HDFS, the DataNode uses a volume-choosing policy to pick a disk for the block. Two such policies that are currently supported are —

1. **Round-robin policy:** The data nodes use the round-robin policy by default. It distributes data blocks in a uniform way across all the available disks.

2. **Available space policy:** It distributes data blocks depending on which disk has the maximum free space.

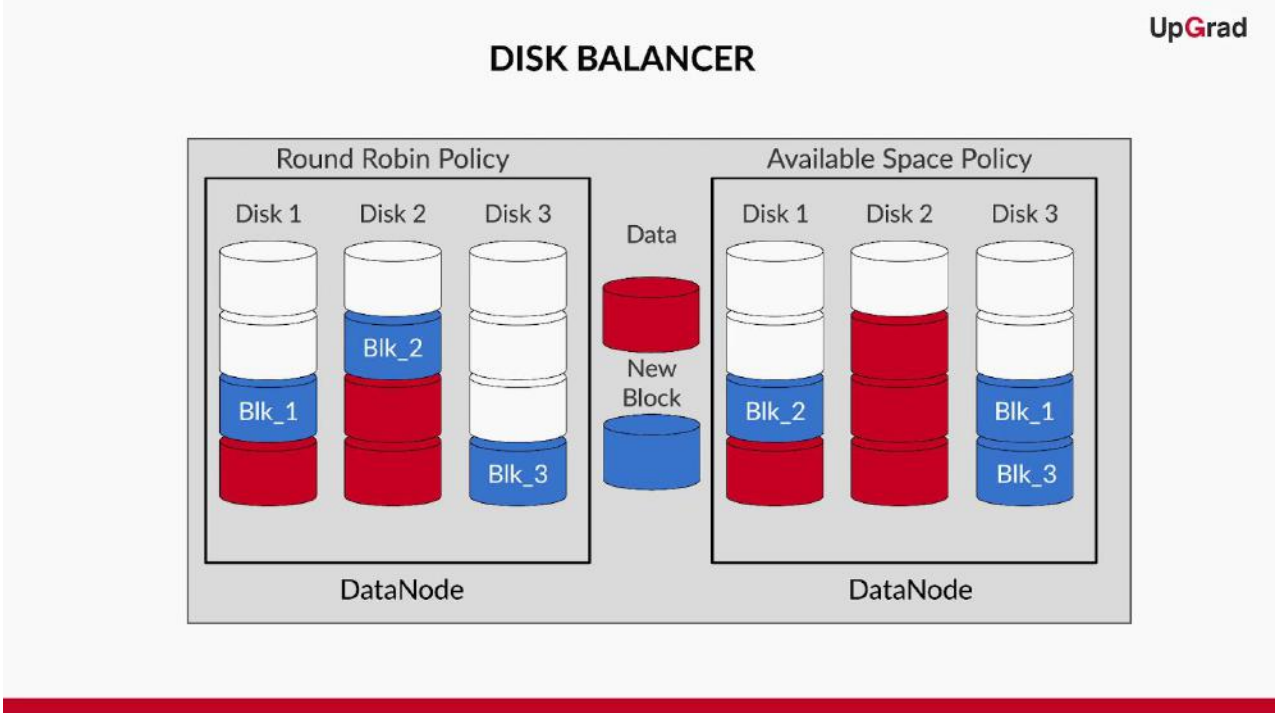


Figure 15: Disk balancer

In a long running cluster, due to massive file deletion and addition operations in the HDFS, it is still possible for a DataNode to create a significant imbalance in the volumes. Even the available space-based volume choosing policy can lead to less efficient disk I/O. For example, consider a situation when every new write operation goes to the newly added empty disk, and other disks in the same machine are idle during this period. This frequent access to the new disk will lead to deadlocks.

The HDFS disk balancer works by creating a plan and executing that plan on the DataNode. The plan determines how much data should move between two disks and includes many '**move steps'** (move steps have data of the source disk, destination disk, and the number of bytes to be moved).

By default, the disk balancer is not enabled. To enable the disk balancer, '**dfs.disk.balancer.enabled**' must be set to true for all data nodes. From CDH 5.8.2 onwards, a user can specify this configuration via the HDFS safety valve snippet in Cloudera Manager. The

Apache Hadoop community has subsequently developed offline server scripts to reduce the data imbalance issue.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Lecture Notes

Distributed Computing Environment for Big Data

This module dealt with distributed systems and their characteristics. It also covered clusters: specific implementations of distributed systems, that are used to handle big data. The second session of the module introduced you to Hadoop clusters and the different components of the Hadoop ecosystem.

What are Distributed Systems?

A distributed system comprises many independent computers connected to each other over a network, running a set of services that makes it appear as a single machine. The computers in a distributed system communicate and coordinate their actions by sending and receiving messages to solve common tasks.

Let's consider an example of this communication and coordination. For Aadhaar identity verification, let's imagine that the biometric checks or biometric processing are performed by one computer, and the demographic credentials are verified by another computer. These two computers must communicate and coordinate to finally authenticate a user. Now, there are around six crore user authentications done per day; huge amounts of computational and storage power are required to perform such a large number of tasks. However, it is all possible using the distributed system we just described.

The database of Aadhaar, which is in petabytes, is stored in parts via multiple servers situated in different locations in India: Bangalore, Bombay, Delhi, Madras, and more. When an authentication request originates, it is moved to the location where the required data is stored. Hence, you can imagine that parallelly, several such authentication checks are carried out at multiple locations, improving the response time.

Let's now see how Aadhaar authentication usually works, with reference to the image given below:

1. First, a user submits his/her Aadhaar details to a vendor who uses a point-of-sale device provided by a bank, such as State Bank of India, to capture the data stored on the user's Aadhaar card.

2. This data is sent to authorisation service agencies, also called ASAs, such as RailTel Corporation, which then forwards the details to the CIDR; this is the Central Identities Data Repository, which is the backbone of the Aadhaar system.

3. The CIDR matches the user's card details with his/her stored details; it then validates the person's identity.

4. The response comes back to the vendor via the same channel of communication.

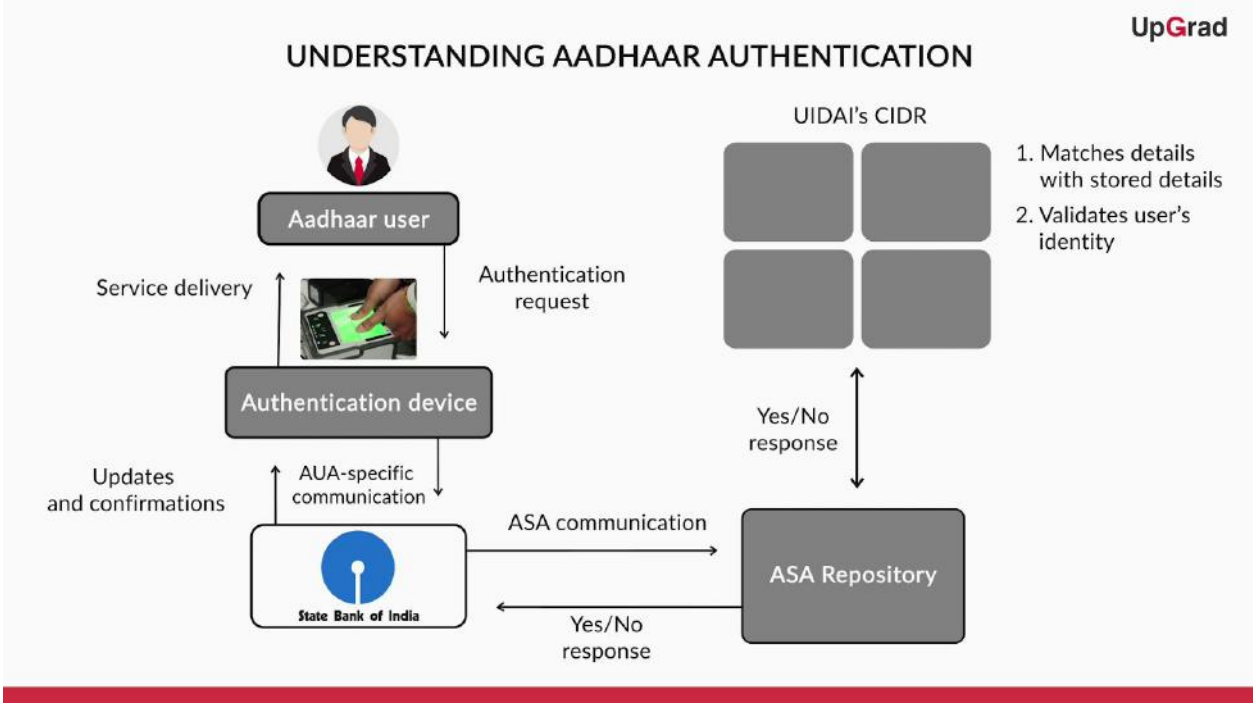


Figure 1: Understanding Aadhaar Authentication

Now, to do all this, Aadhaar uses the Hadoop framework for storage and processing. This framework includes several software frameworks such as the Hadoop Distributed File System (HDFS), MapReduce, Spark, Tez, etc.

Properties of Distributed Systems

The four significant characteristics of distributed systems are —

1. **High availability:** Distributed systems ensure availability, i.e. even if a component of a system fails, the system doesn't fail, and there is another component that replaces it and keeps it running.
2. **Scalability:** To ensure that it doesn't fail, a distributed system is flexible so that you can easily increase the number of machines when the workload increases.
3. **Transparency:** Distributed systems provide a global view of all the underlying machines as a single machine and hide their internal workings from end users.
4. **Flexibility:** Due to advancements in the field of microprocessors, networking, and storage, distributed systems have made it possible to make changes to both hardware and software components, without affecting performance.

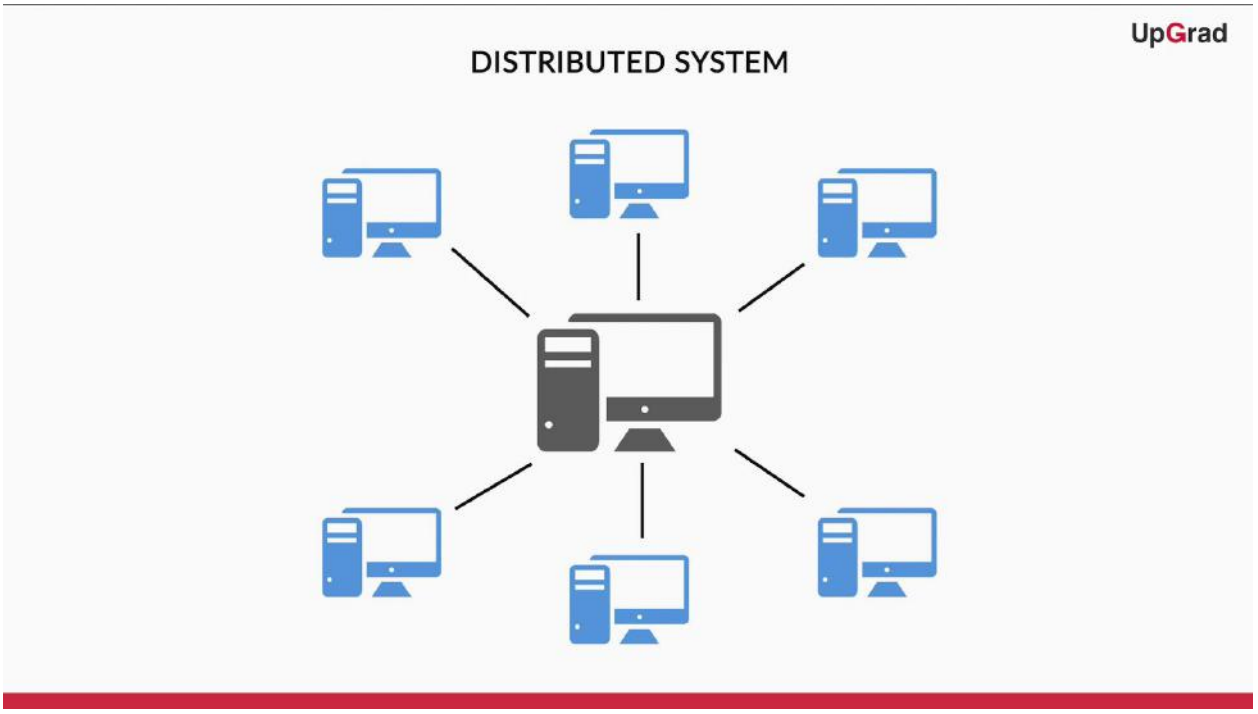


Figure 2: Distributed System

Clusters

A cluster is defined as a group of computing machines (also called nodes) working together and configured in such a way that they appear as a single system. Clusters use several low-cost commodity machines that are connected through a communication network. The image on the next page illustrates a schematic of the cluster architecture**:**

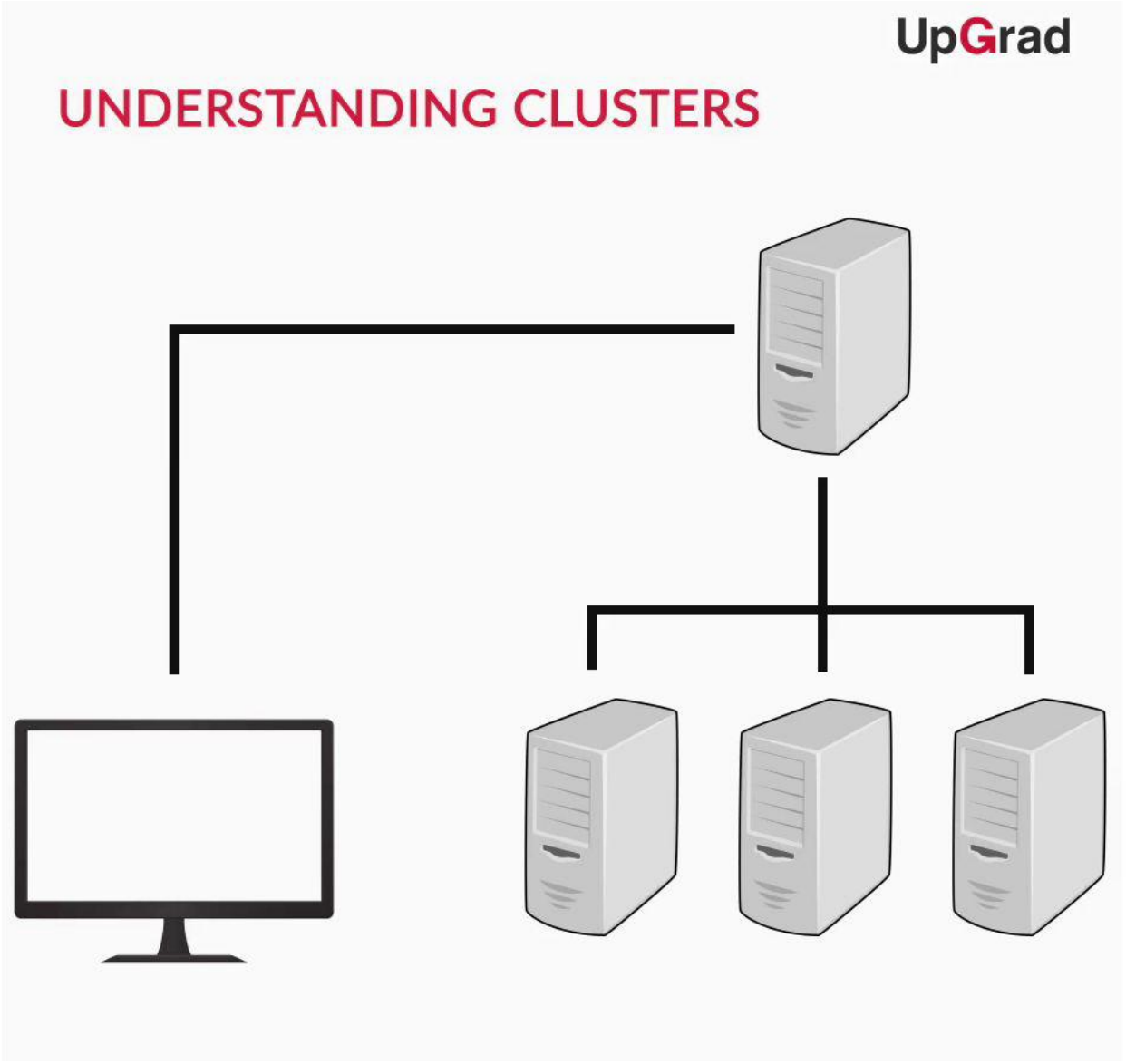


Figure 3: Cluster Architecture

Google File System

Developers at Google used to deal with large files that were difficult to manipulate using standalone computing machinery on a regular basis, due to the sizes of the files. The overall network in which these files existed was also huge. Monitoring and maintaining the files was also challenging. There was a need to automate the administrative duties required to keep the system running.

One of the key design decisions made for the system was the concept of simplification. It was concluded that as systems grow more complex, problems arise more often. Hence, a simple approach is easier to control even when the scale of a system is huge. Based on this, it was decided that users would only have access to basic file commands such as open, read, write, and close.

Google Search Engine

The files in GFS were very large: in the range of a few gigabytes to terabytes. Therefore, an important factor to consider while distributing these files across the system was the network cost, both in terms of time and money. To address this problem, GFS now breaks files into chunks of 64 megabytes each. Each chunk or block is provided a unique 64-bit identification number called a 'chunk handle'.

Google organised its GFS into two clusters of computers, each containing thousands of machines. As shown in the image given below, each GFS cluster has three entities, namely —

1. **Client:** The entity that sends file requests to both the master and chunk servers.
2. **Master server:** The coordinator of the cluster. It maintains an operation log that keeps track of the activities of the particular master's cluster. It also keeps track of metadata, which is information that describes the chunks or files that have been broken down and stored on multiple chunk servers.
3. **Chunk server:** A machine that is responsible for storing 64-megabyte file chunks.

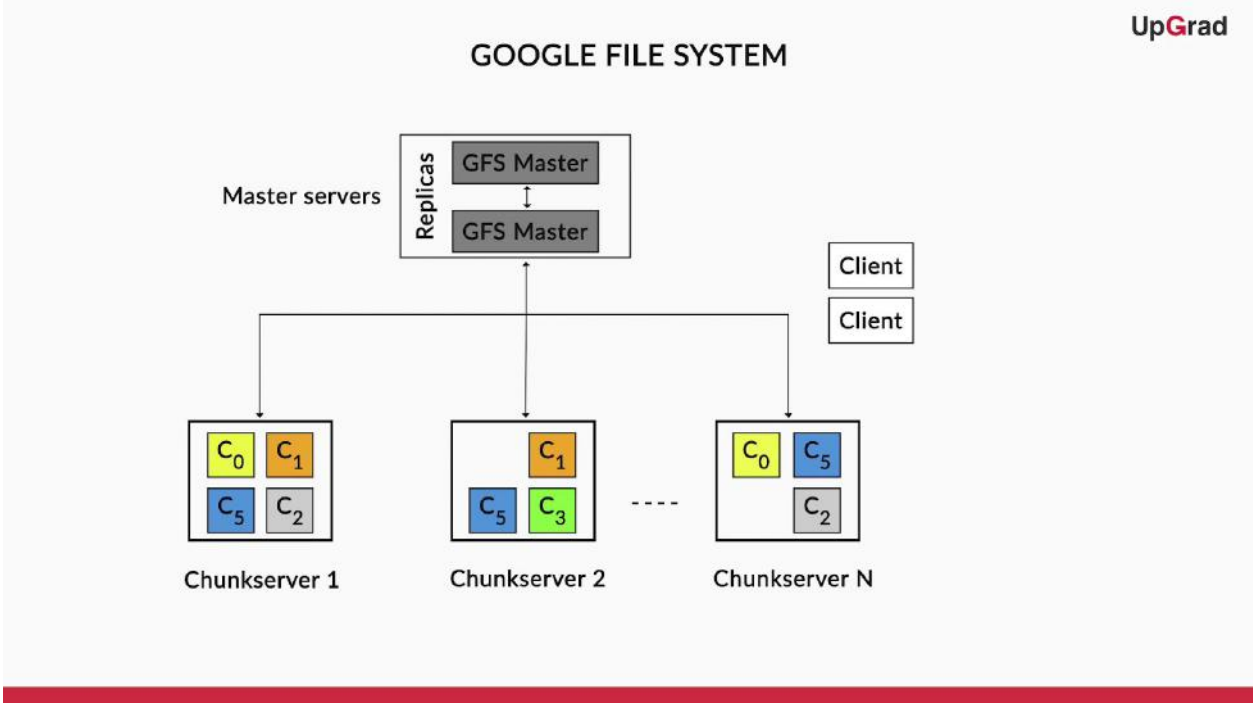


Figure 4: GFS Cluster

Even though there are multiple copies of the master servers in a cluster, there is only one active master server per cluster. To prevent data loss, GFS replicates every chunk multiple times and stores the replicated chunks on different chunk servers. These chunks are known as replicas.

Read request in GFS

1. The client sends a request to the master server to find out where it can find a particular file in the system.
2. The server responds with the location of the primary replica of the respective chunk.

Write request in GFS

1. The client sends a request to the master server.

2. The master server then sends the locations of the primary and secondary replicas back to the client.

3. The client stores this information in the memory cache so that if it needs to refer to the same replica later, it can bypass the master server and get the replica from the cache itself. The write data is then sent to all the replicas by the clients, starting with the closest replica and ending with the farthest. Whether the replicas are primary or secondary does not matter here.

4. The primary replica assigns consecutive serial numbers to each change in the file once it receives the data post the write request. These changes are called mutations, and the serial numbers help the replicas to define an order for the mutations.

5. The primary replica then applies the mutations in sequential order to its own data, post which it sends a write request to the secondary replicas, which follow the same application process.

6. On the completion of this process across all the replicas in the cluster, the secondary replica reports back to the primary replica, which, in turn, reports back to the client.

7. If successful, the process ends. If not, the primary replica passes an error message to the client.

This is called a 'pipeline right', where once a right is completed at one chunk server, it is forwarded to the next chunk server which keeps a replica, and so on. If a secondary replica does not update correctly, a primary replica tells it to start all over again. If this also fails, the master server identifies the same secondary replica as garbage.

Some of the added functionalities of GFS are —

1. Replicas of the master server exist. They monitor the operations log, regularly communicate with the active master, and take over the active master server in the case of failure.
2. Chunks are replicated in such a way that they ensure data availability even in the case of hardware failures. Replicas are stored on different machines across different racks.

3. The master server monitors the cluster and helps in balancing the load across it. It also ensures that the chunk servers run at near-capacity but never at full capacity.

On average, a single query on Google reads a hundred megabytes of data and consumes tens of billions of CPU cycles. To meet the user requirements of fast search responses, Google uses computing clusters to reply to queries that come from clients.

Processing mechanism of a search query by Google's search engine

Google has many search clusters spread across the globe. When a user's query is received, the company directs it to the nearest cluster hosting the Google web server, index servers, and document servers. The web server then executes the task in two phases, using the index and document servers:

First phase: The index servers consult an inverted index that maps each query word to a matching list of documents called the 'hit list'. Then, they determine a set of relevant or matching documents by intersecting this hit list of individual query words and computing a relevance score for each document. The final result of this first phase of query execution is an ordered list of document identifiers, or, in short, doc IDs.

Second phase: This phase involves taking the ordered list of doc IDs from the first phase and finding the IDs' titles and uniform resource locators (URLs), along with query-specific document summaries using document servers.

The Google cluster also performs additional tasks such as spell checks and, last but not least, serving relevant advertisements.

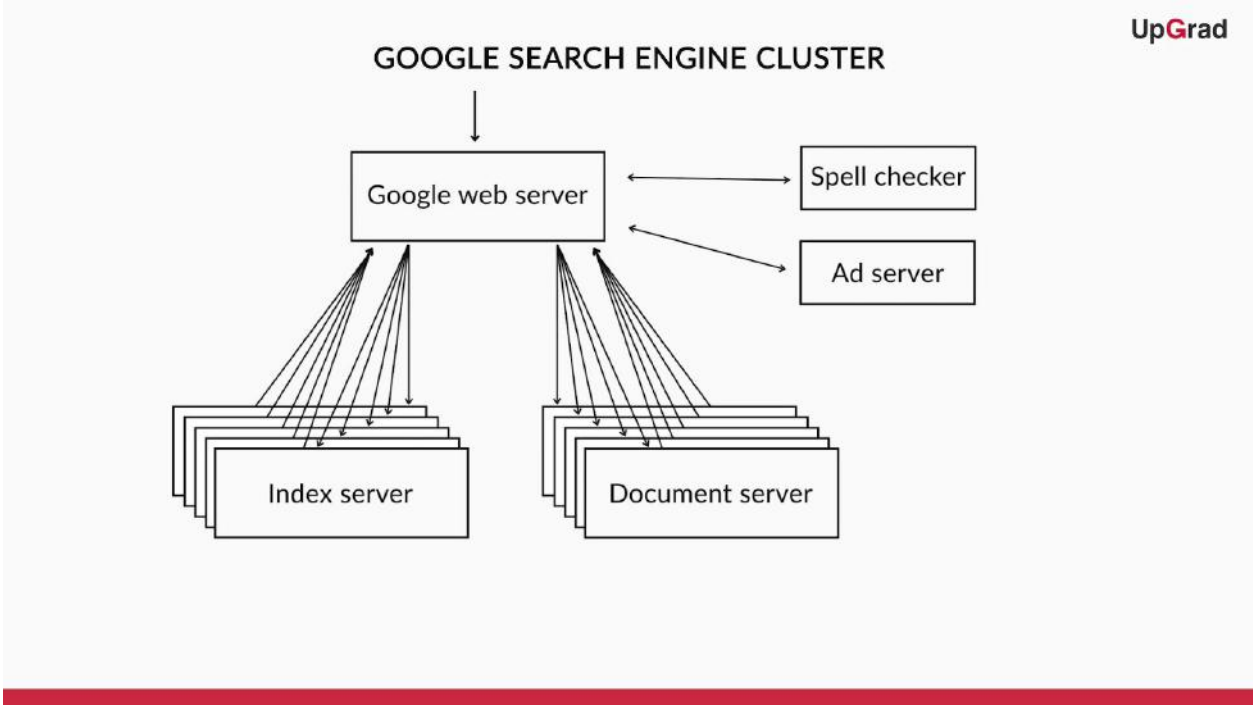


Figure 5: Google Search Engine Cluster

Hadoop Framework

Hadoop is a software framework that helps in storing and processing big data efficiently. It partitions large datasets into smaller chunks and performs parallel computations on them. The storage and computational capabilities of Hadoop increase with the addition of new nodes to a cluster. The two primary components of a Hadoop distributed master-slave architecture are —

1. Hadoop Distributed File System (HDFS)
2. MapReduce

Hadoop also follows this master-slave architecture. The following image is a schematic of the Hadoop architecture:

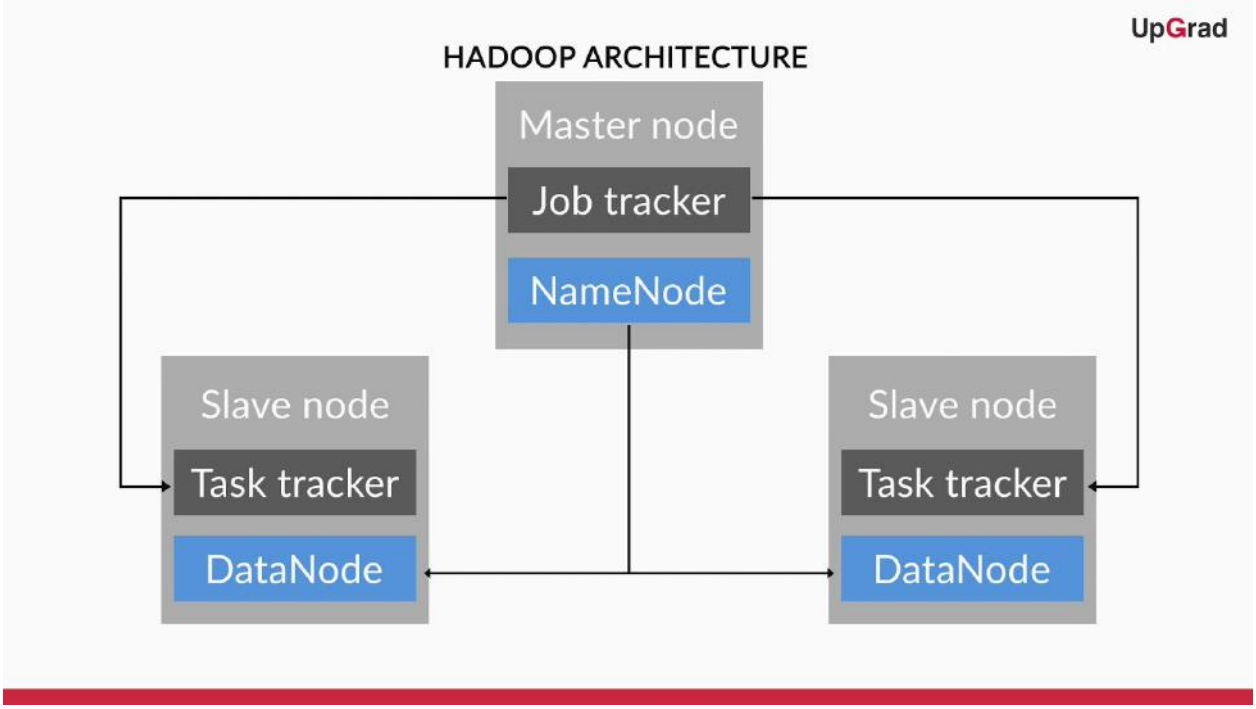


Figure 6: Hadoop Architecture

The HDFS master (NameNode) partitions the large amount of data that is to be stored in the HDFS into smaller chunks and stores it across the slave nodes (DataNodes). It also maintains the metadata about where the data or chunks are located. In other words, it works as a directory service. The MapReduce master (job tracker) is responsible for deciding where computational work is scheduled on slave nodes.

Hadoop Distributed File System(HDFS)

The Hadoop Distributed File System (HDFS) is the backbone of the Hadoop ecosystem and provides a platform to store different types of large datasets on data nodes in the form of data blocks, along with the

metadata required to locate these blocks on the NameNode. As shown in the image given below, the HDFS architecture has three entities, namely —

1. **Client:** HDFS clients interact with the NameNode for metadata about files and with data nodes for the actual reading and writing of files.
2. **NameNode:** The HDFS NameNode maintains the metadata of the file system, such as which data node has which block of a file.
3. **DataNode:** Files are made up of blocks, and each block can be replicated multiple times and can be stored on multiple data nodes. Data nodes communicate with each other, pipelining these rides.

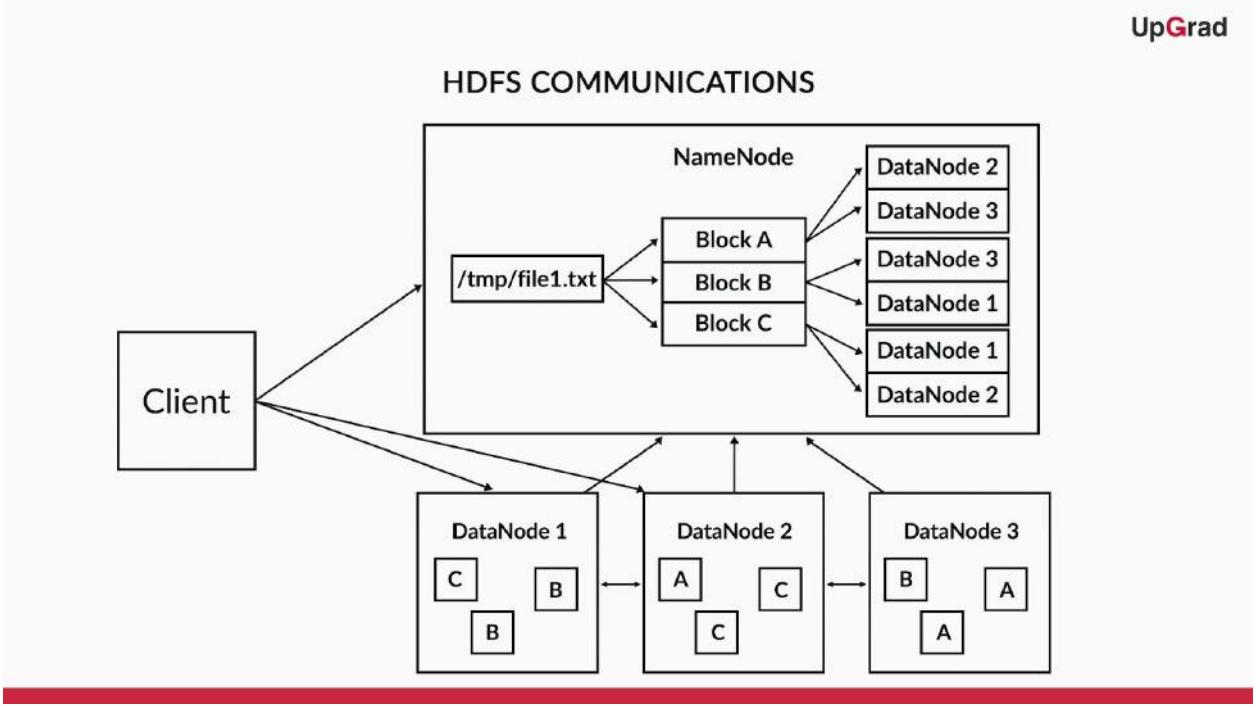


Figure 7: HDFS Architecture

Some of the features of the HDFS are —

1. **Data isolation and high availability:** The HDFS allows the sharing of NameNode content (metadata) among NameNodes within a single cluster, thereby allowing data isolation and high availability.
2. **Data locality and fault tolerance**: The system divides data into large blocks, known as chunks, and moves the computation towards the data, thereby providing data locality. Further, file chunks are replicated to provide fault tolerance.
3. **High throughput:** The HDFS being a system of distributed storage that works optimally for large files, it provides high throughput while dealing with large volumes of data.
4. **Transparency**: The system creates a level of abstraction over the distributed data blocks from where you can visualise the entire file system as a single database or unit.

MapReduce

MapReduce is a batch-oriented distributed computing framework that utilises the data-parallel programming model. The workflow of the MapReduce framework is shown in the image given below and may be summarised as follows:

1. The MapReduce framework breaks a large task into small Map and Reduce tasks, and these Map and Reduce tasks are run parallelly on the data nodes that store input data.
2. Each Map function takes a key-value pair that represents a record from the input data source, and produces some intermediate key-value pairs as output.
3. These intermediate key-value pairs become the input for the Reduce tasks, which then give the final output after performing the Reduce function on them.

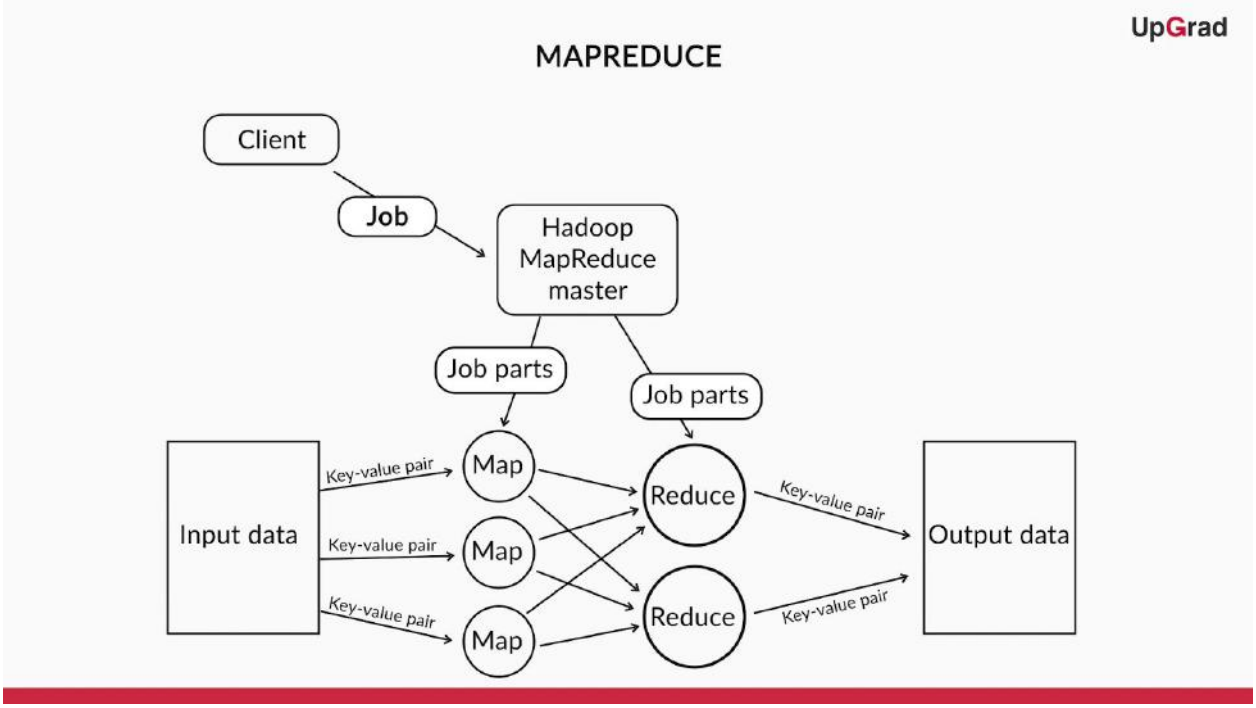


Figure 8: Workflow of MapReduce Framework

To understand the workflow in more detail, consider an example of computing the number of students in each department of an engineering institute. Given the data of students and their respective disciplines (departments), input this data to a map function. This mapper will output their respective key value pairs (key - department, value - count of students). Now these mapper outputs will act as the input to the reducers, which will combine the key values for the same key (same department) as shown in the image given below.

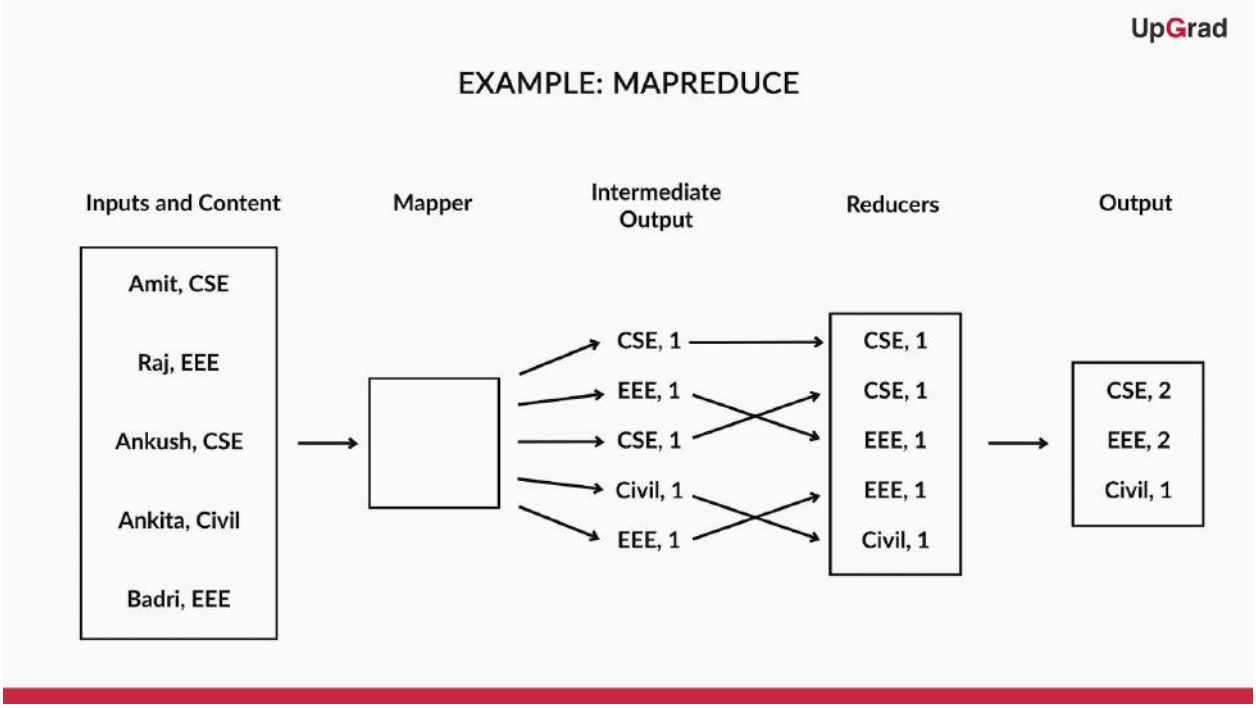


Figure 9: Example 1

Consider another example of files that contain different sets of words. To compute on the list of files in which a particular word belongs to, give these files as inputs to mappers. These mappers will take each of the words and output a key value pair (where key is the particular word and the value is the document in which this particular word exists). Now these key value pairs which are the intermediate output from the mappers are passed through two phases, called shuffle and sort. These shuffle and sort phases are responsible for determining the reducer that should receive the map output key value pair and for sorting all the input keys for a given reducer. The reducer then collects the file names for each key (word) and outputs the records which will contain the keys (word) and the list of files in which these particular keys (word) are present as shown in the image given below.

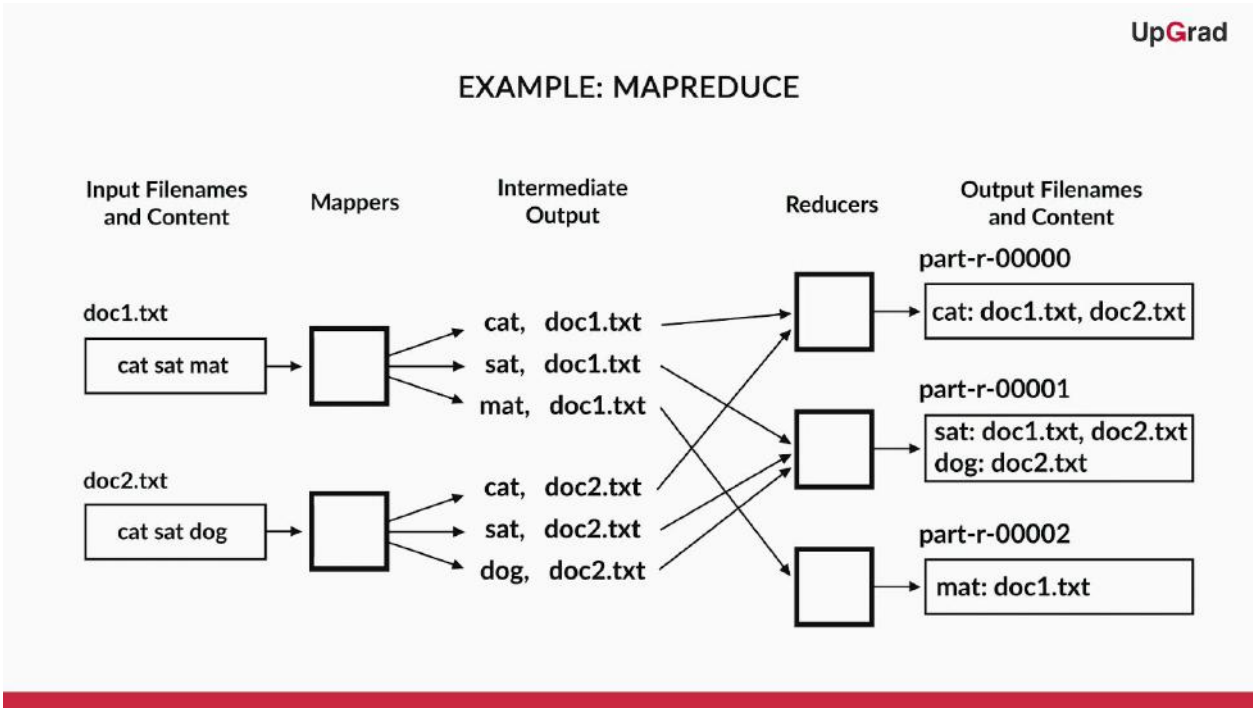


Figure 10: Example 2

Hadoop Ecosystem

Hadoop is a framework that provides a suite of services to solve big data problems efficiently. It includes several components that perform specific tasks.

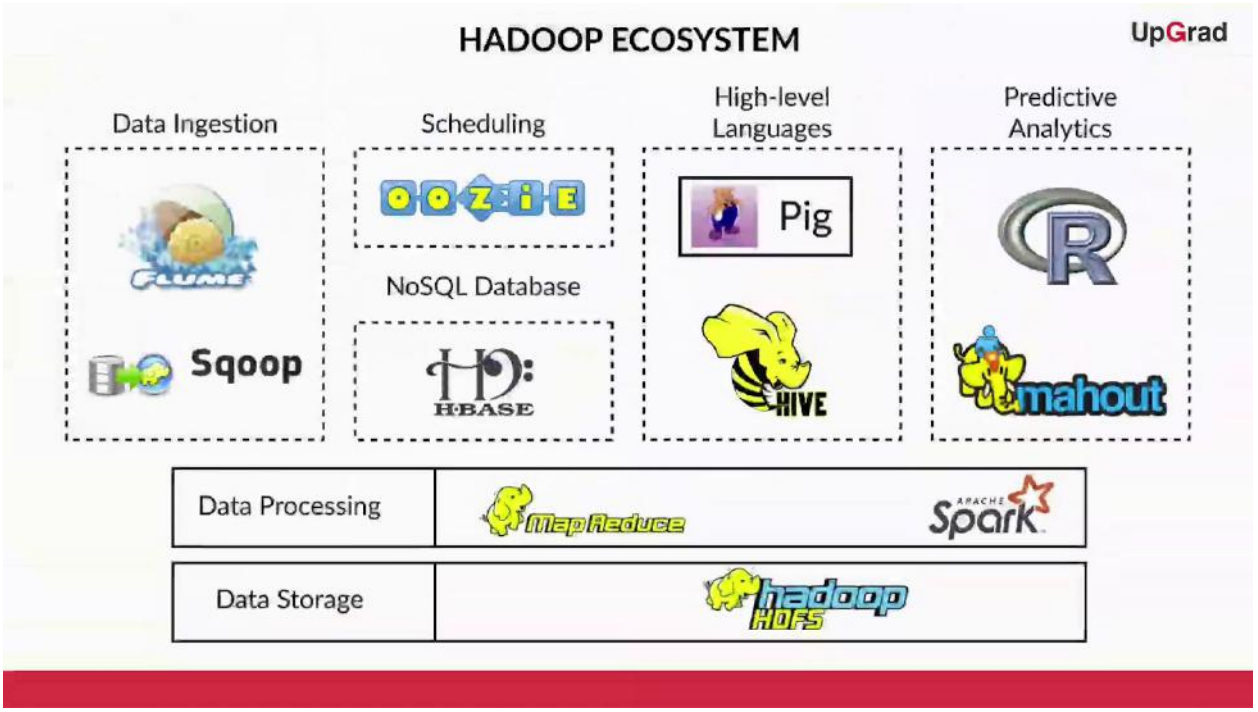


Figure 11: Hadoop Ecosystem

Some commonly used components of the Hadoop stack are —

1. **Pig:** A high-level language for writing programs that run on Apache Hadoop, Pig allows user-defined functions, thereby making programming much simpler than writing MapReduce codes in Java.
2. **HBase:** A column-based NoSQL database that provides high throughput and low latency on Hadoop platforms, HBase offers a fault-tolerant method to store large quantities of sparse data and perform real-time analytics.
3. **Sqoop:** Sqoop is an application that provides a command-line interface for transferring data between Hadoop and relational databases.
4. **Hive:** Hive is a data warehouse that provides an SQL-like interface to query databases and file systems that are integrated with Hadoop.
5. **Flume:** This is a distributed framework that efficiently collects, aggregates, and moves large amounts of unstructured and semi-structured streaming data into the HDFS.
6. **R:** R is a programming language used to derive insights from big data. It is used for a variety of statistical techniques, including linear and non-linear modelling, time series analysis, classification, clustering, etc., for mining information from raw data. The commonly used methods of integrating Hadoop with R are RHadoop and RHive.
7. **Apache Mahout:** Mahout provides implementations of distributed and scalable machine learning algorithms focused on collaborative filtering, clustering, and classification.
8. **Oozie:** Oozie is a workflow management system that helps in defining and monitoring a sequence of tasks on a Hadoop cluster.
9. **Apache Spark:** Spark is a framework for real-time data analytics in a distributed computing environment, which increases the speed of data processing over MapReduce frameworks by executing computations in memory. This extends the MapReduce model to efficiently use more types of computations, which include iterative queries and stream processing.

Amazon DynamoDB Lecture Notes

In the previous sessions of this course, you learnt about HBase, which is a columnar-type NoSQL datastore service offered by Apache.

In this module, you were introduced to DynamoDB, which is NoSQL database service offered by Amazon AWS and supports both key-value data store and document data store.

Fundamentals of NoSQL Databases

You learnt that unlike relational database, where there is a predefined schema, NoSQL database supports a schemaless architecture, and the design of the database can be changed very easily. There are primarily four different types of NoSQL database architectures:

1. **Key-value data store**
2. **Column-oriented data store**
3. **Document data store**
4. **Graph data store**

In document data stores, a collection of values describe a certain item. The values are unique to a specific item and are compressed together to form a document describing or representing that item. The values stored in the document usually follow a specific industry-relevant format, such as JSON, XML, or BSON.

A relational database supports a schema-on-write operation, where the database has a predefined schema and the data added to this database conforms to this predefined structure, as shown in the table below:

Table 1: A Sample Relational Table

Employee ID	Employee Name	Age	City	Salary
1	Shiva	26	Hyderabad	50,000
2	Karanpreet	25	Mumbai	75,000
3	Salma	22	Mumbai	60,000
4	Jeanona	33	Mumbai	1,00,000

A NoSQL database, on the other hand, doesn't require any predefined schema. This means unstructured data can be easily added to a NoSQL database. As you can see in the table below, different types of products with different components can easily be stored using a single table.

Table 2: A Sample NoSQL Table

Product Category	Product Type	Contents
Electronics	Mobile	Model No.: iPhone X Storage: 64 GB Colour: Space Grey Camera: 12 MP
Electronics	Laptop	Model No.: Dell Vostro Storage: 1 TB OS: Windows 10 RAM: 8 GB
Home Appliances	Microwave	Manufacturer: LG Mode No.: LG-HMW Capacity: 25 L Colour: White
Home Appliances	Television	Manufacturer: Samsung Type: LED Resolution: 4K Size: 19 Inches
Entertainment	Movie	Title: Titanic Cast: Leonardo Di Caprio, Kate Winslet Director: James Cameroon IMDb Rating: 7.8/10

Basic Features of DynamoDB

In DynamoDB, you learnt that data is stored in the form of key-value pairs. A sample table shown below, represents a key-value pair relationship:

Table 3: A Key-Value Pair Example

Key	Value
Country	India
State	Maharashtra
City	Mumbai

Pin code	400018

Every row in such a table is called an item, where each item is made of a key-value pair. The key declared should be unique and should have a value. The key can be a string, a number, or a binary, and the value can be either single-valued or a multivalued set, e.g. JSON or BLOB (Basic Large Object).

Furthermore, the various key features of DynamoDB are as follows:

1. **Auto Scaling**: DynamoDB **scales up** the platform capacities automatically when the databases experience heavy user traffic or a sudden spike in the number of users.

2. **High availability**: DynamoDB **automatically replicates data** stored in the database across the availability zones. This process ensures **high availability** and no loss of data even if one of the zones loses the data because of individual machine failure.

3. **Self partitioning:** DynamoDB allocates additional partitions to a table if the table's provisioned throughput capacities are increased beyond what the existing partitions can support, or if an existing partition fills to its capacity and more storage space is required. Note that partition management occurs automatically in the background and does not require any manual intervention.

4. **Performance monitor**: DynamoDB provides tools to monitor the performance of the database by letting the user compare the consumption of the read-and-write capacity units against the provisioned/set throughput capacities. This feature helps the user **scale down the capacities** in case less units are being utilized, in order to **save on cost**.

Core Components of DynamoDB

In this module, you were introduced to various core components of a DynamoDB table, which included items, attributes, partition keys, etc. We discussed in detail the significance and functioning of each component of a table.

The core components of DynamoDB are as follows:

- **Table**: As in RDBMS systems, DynamoDB stores the data in the form of tables. A table in DynamoDB is a **collection of items**, and every table should have a **unique name** for a specific account and region.

- **Item:** Data in a table is stored in the form of items, where each row represents a unique item. The maximum size of an item in a table can be **400 KB.** In the session videos, you learnt that each row,

i.e. each order in the sample e-commerce database we created in the video, which was stored in a specific row, served as an item in the table.

- **Primary key (hash key)**: This is a mandatory attribute defined at the time of creating a table. Each primary key should have a **unique value**, and no two items can have the same primary key value in a table, unless we have declared an optional sort key. In the table show below, **CustomerID** can operate as a primary key and **OrderID** as a sort key, to give the desired item.

Table 4: NoSQL Database

Items	**Primary key**	**Sort key**	**Attributes**
	CustomerID	**OrderID**	
Item 1	101	Order001	Model No.: iPhone X Storage: 64 GB Colour: Space Grey Camera: 12 MP
Item 2	101	Order007	Model No.: Dell Vostro Storage: 1 TB OS: Windows 10 RAM: 8 GB
Item 3	102	Order111	Title: Titanic Cast: Leonardo Di Caprio, Kate Winslet Director: James Cameroon IMDb Rating: 7.8/10

Components of DynamoDB Table

- **Sort key (range key)**: This is an optional attribute declared while creating the table. However, declaring the sort key helps in narrowing down the search results while querying a particular set of data.

- **Attribute:** An attribute can be considered a property associated with an item. Each item can have one or more attributes, and it is the **fundamental element of data**, which can't be broken down further.

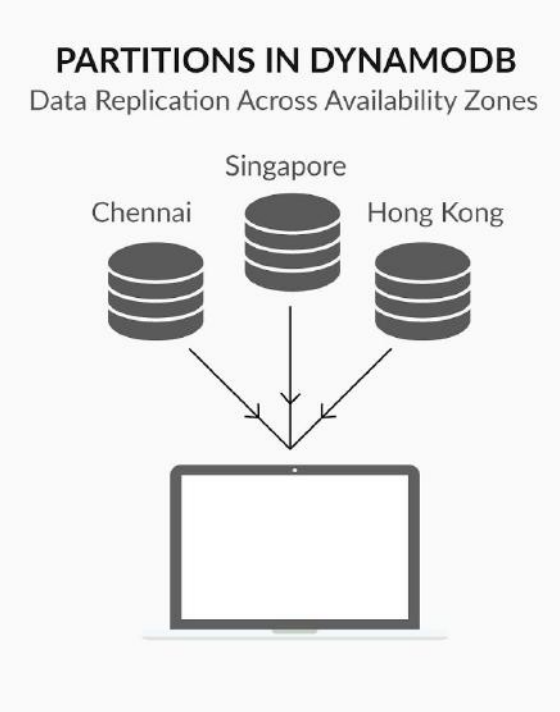
Partitions & Composite Key

In the previously explained concepts, you learnt about various core components, such as tables, items, attributes, etc., of a DynamoDB table. Now let's summarize and learn all about partitions and composite keys in a DynamoDB table

Partitions:

- An important feature of DynamoDB is the way data is stored. Data in DynamoDB is stored in partitions. All the data written into any table is stored in a specific partition, which is an allocation of storage for a table and is backed by **solid-state drives (SSDs).**

- The partition is automatically replicated across multiple **availability zones** within an AWS region to ensure complete availability of data in the table. This ensures that if one availability zone is down, or has lost any data, then that data can be easily fetched from another zone.



- In order to decide which data gets allocated to which partition, DynamoDB makes use of the partition key.

- Partition management is handled entirely by DynamoDB; you never have to manage partitions by yourself.

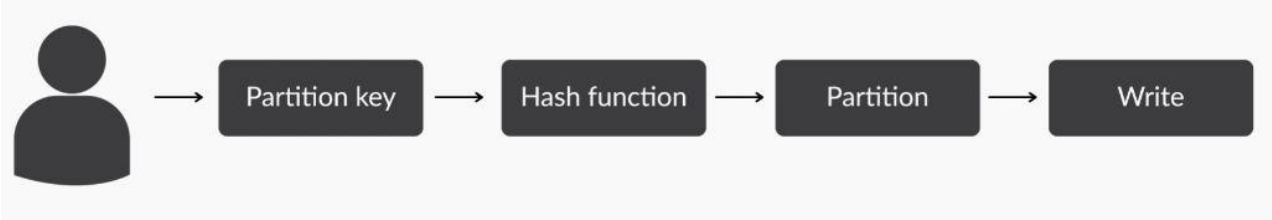
- DynamoDB makes use of the **partition key** to allocate any data to a specific partition.

- Partition allocation is completely a **self-managed** process handled by DynamoDB.

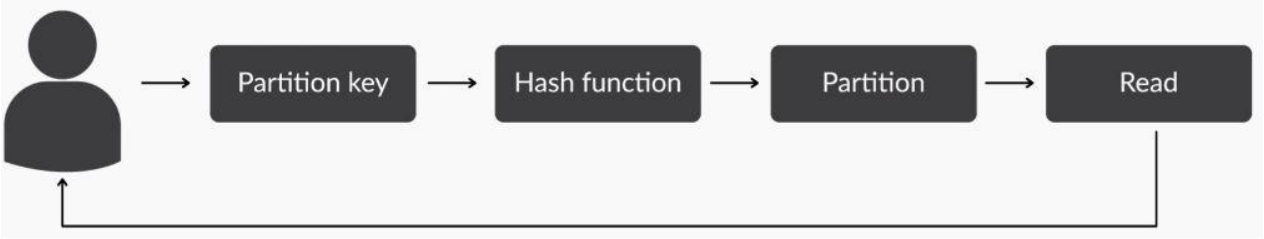
Partition Key:

- Each item getting stored in DynamoDB is required to have a unique primary key, also known as the partition key. This partition key functions as a **unique key identifier** to get access to the desired item.

- When you write an item into the database, the partition for that specific item is decided on the basis of the **partition key value**. DynamoDB passes the partition key as an input to an **input hash function.** The output from this hash function determines the partition in which the item will be stored.



- As in the case of reading an item from the table, DynamoDB uses the partition key value as input to its hash function, yielding the partition in which the item can be found.



Composite Key (Partition and Sort Keys)

- A combination of the partition and sort keys is commonly known as a composite key. Sometimes data items can be segregated on a broad level using a partition key, but they still require an additional key, i.e., a sort key to uniquely identify the data item.
- As mentioned earlier, declaring the sort key helps a user in narrowing down the search results while querying a particular type of data.

Create a DynamoDB Table

In this exercise, you learnt how to create a DynamoDB table by writing a Java program. The name of the table is 'CustomerData', and it contains the data of all the orders placed by some customers on an e-commerce website. It is made up of primary and sort keys, which are 'CustomerID' and 'OrderID', respectively.

The code for creating a table is shown below:

```
package com.amazonaws.samples;
import java.util.Arrays;
import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Table;
import com.amazonaws.services.dynamodbv2.model.AttributeDefinition;
import com.amazonaws.services.dynamodbv2.model.KeySchemaElement;
import com.amazonaws.services.dynamodbv2.model.KeyType;
import com.amazonaws.services.dynamodbv2.model.ProvisionedThroughput;
import com.amazonaws.services.dynamodbv2.model.ScalarAttributeType;

public class Customerdata {

   public static void main(String[] args) throws Exception {

      AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
                                                          .withEndpointConfiguration(new
AwsClientBuilder.EndpointConfiguration("https://dynamodb.us-west-2.amazonaws.com", "us-west-2"))
          .build();

      DynamoDB dynamoDB = new DynamoDB(client);
      String tableName = "CustomerData";
      try {
         System.out.println("Attempting to create table; please wait...");
         Table table = dynamoDB.createTable(tableName,
```

```
            Arrays.asList(new KeySchemaElement("CustomerID", KeyType.HASH), // Partition Key
               new KeySchemaElement("OrderID", KeyType.RANGE)), // Sort key
            Arrays.asList(new AttributeDefinition("CustomerID", ScalarAttributeType.S),

               new AttributeDefinition("OrderID", ScalarAttributeType.S)),
            new ProvisionedThroughput(10L, 10L));
        table.waitForActive();
        System.out.println("Success.  Table status: " + table.getDescription().getTableStatus());

      }
      catch (Exception e) {
          System.err.println("Unable to create table: ");
          System.err.println(e.getMessage());
      }

   }
}
```

In the code, it is shown that the first statement calls the java.utils.Arrays package. Along with this, a few more packages need to be called to create a table. All the packages starting with com.amazonaws are used to program DynamoDB.

Next, you will have to create a class, which in this case is 'Customerdata'. Inside the main method of this class, you will have to create an instance of AmazonDynamoDB class, which in this case is 'client'. This instance is used to connect to a data center using the withEndpointConfiguration() method. This means that you need to specify a URL of the data center where the request should be sent and from where the response should be received. In this case, the data center is Oregon (West), which is one of the AWS regions. The URL for Oregon (West) is https://dynamodb.us-west-2.amazonaws.com. Here, us-west-2 refers to the region Oregon (West).

To access the list of all the AWS regions, click here. You can choose any data center or AWS region you wish, but stick only to one; do not change the AWS regions.

Next, you need to create an instance of the class DynamoDB, which in this case is 'dynamoDB'. Its constructor takes 'client' as the input parameter. The 'client' variable contains all the required information of the data center situated in Oregon (West).

You now need to declare a string type variable called 'tableName', which will store the name of the table. In this example, the name of the table is 'CustomerData'.

Next, in the try block, you need to create an instance of the 'Table' class with the help of the dynamoDB instance. While creating the instance, use the createTable() method to create the required table. The

input parameter for the createTable() method is the tableName variable, which was declared earlier. Along with tableName, you need to pass the Arrays.asList() method, which indicates that a table is a list of items.

Inside the Arrays.asList() method, you need to pass the KeySchemaElement() method, which is used to declare the primary and sort keys. The keywords to distinguish between primary and sort keys are KeyType.HASH and KeyType.RANGE respectively. Furthermore, you need to specify the attribute type of the primary and sort keys. To do this, you need to use the keyword ScalarAttributeType. You already know that both the keys can have only scalar attributes, which are string, or number, or binary. The word scalar means that the attribute value will be a single value, not a multi-value.

ScalarAttributeType.S indicates that the attribute type for the key is a scalar string.
ScalarAttributeType.N indicates that the attribute type for the key is a scalar number.
ScalarAttributeType.B indicates that the attribute type for the key is a scalar binary.

To define the attribute type, you need to use the AttributeDefinition() method.

Finally, you need to finish the Arrays.asList() method by specifying the throughput values using the ProvisionedThroughput() method. Throughput values indicate the speed at which data is read from the table and the speed at which data is entered in a table.

To measure the reading speed, the RCU metric is used, which stands for Read Capacity Units, and to measure the writing speed, the WCU metric is used, which stands for Write Capacity Units. In this example, the value for both RCU and WCU is 10, which means you can read 10 units of the table per unit time and can write 10 units in the table per unit time.

If you wish to create a table by writing a Java program, you will have to specify the throughput values mandatorily.

Finally, you can finish the try block by writing the table.waitForActive() statement. This statement means that DynamoDB is creating the table requested by the user but is not yet ready to use. Once the status of table turns to ‘active’, which means that the table is successfully created, you can add data into the table or view the table.
Inside the System.out.println() method, the getDescription() and getTableStatus() methods can be used to get the table details.

Loading Data into a Table

To add data into the table, the following code is used:

```
package com.amazonaws.samples;


import java.io.File;
// Imported for reading, writing on other files and other operations.


import java.util.Iterator;
//This package needs to be installed to iterate over different data structure elements like array, list and all.


import com.amazonaws.client.builder.AwsClientBuilder;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDB;
import com.amazonaws.services.dynamodbv2.AmazonDynamoDBClientBuilder;
import com.amazonaws.services.dynamodbv2.document.DynamoDB;
import com.amazonaws.services.dynamodbv2.document.Item;
// This package is imported to load items in DynamoDB tables.


import com.amazonaws.services.dynamodbv2.document.Table;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
// JsonParser contains methods to parse JSON data using the streaming model
// JsonParser provides forward, read-only access to JSON data using the pull parsing programming model.
// In this model, the application code controls the thread and calls methods in the parser interface to move the parser forward or to obtain JSON data from the current state of the parser


import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;
// Jackson is an open source library to process JSON.
```

```
public class LoadData {

   public static void main(String[] args) throws Exception {

      AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
           // Creates new instance of builder with all defaults set.
                                                          .withEndpointConfiguration(new
AwsClientBuilder.EndpointConfiguration("https://dynamodb.us-west-2.amazonaws.com", "us-west-2"))
          .build();

      DynamoDB dynamoDB = new DynamoDB(client);

      Table table = dynamoDB.getTable("CustomerData");

      JsonParser parser = new JsonFactory().createParser(new File("customerdata.json"));

      JsonNode rootNode = new ObjectMapper().readTree(parser);
      Iterator<JsonNode> iter = rootNode.iterator();

      ObjectNode currentNode;

      while (iter.hasNext()) {
         currentNode = (ObjectNode) iter.next();

         String customerId = currentNode.path("CustomerID").asText();
         String orderId = currentNode.path("OrderID").asText();

         try {
```

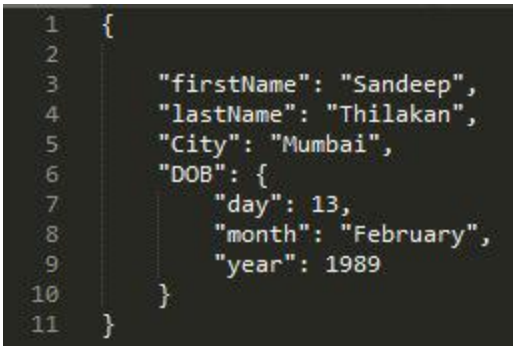
```
                    table.putItem(new Item().withPrimaryKey("CustomerID", customerId, "OrderID",
orderId).withJSON("info",
                currentNode.path("info").toString()));
            System.out.println("PutItem succeeded: " + customerId + " " + orderId);
            // Putting new item in the table


          }
          catch (Exception e) {
            System.err.println("Unable to add item: " + customerId + " " + orderId);
            System.err.println(e.getMessage());
            break;
          }
        }
        parser.close();
    }
}
```

To execute the code above, a new set of libraries needs to be imported in the code, which is as follows:

```
import com.amazonaws.services.dynamodbv2.document.Table;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ObjectNode;
```

These libraries are used to parse a JSON file.

A JSON file is like a TSV or a CSV file, which stores data in the form of objects, or, to be precise, in the form of key-value pairs. The image shown below demonstrates how data is stored in a JSON file in the form of key-value pairs.



As it is shown in the image, “firstName”, “lastName”, and “City” are keys and “Sandeep”, “Thilakan”, and “Mumbai” are their corresponding values. “DOB” is also a key whose value, i.e. {“day”: 13, “month”: “February”, “year”: 1989}, is a list of key-value pairs.

The keys are always written within double quotes followed by a colon. The values can be strings, numbers, lists, arrays, binary numbers, or boolean. String type values are always written within double quotes, list type values are enclosed within curly brackets, or ‘{}’, such that each bracket-separated entity inside the list is separated by a comma, and array type values are enclosed within square brackets, or ‘[]’, such that each bracket-separated entity inside an array is also separated by a comma.

Coming back to the code, the first few lines of the code are going to remain to the same, which are as follows:

```
AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()

.withEndpointConfiguration(new
AwsClientBuilder.EndpointConfiguration("https://dynamodb.us-west-2.amazonaws.com", "us-west-2"))
        .build();

DynamoDB dynamoDB = new DynamoDB(client);
```

The code snippet above creates a DynamoDB client and establishes the connection between Eclipse and one of the AWS regions. In that region, DynamoDB will create the table. The following link lists all the AWS regions.

AWS Regions

Now, a user needs to provide the name of the table that needs to be populated through the JSON file. In this example, a table called ‘CustomerData’ needs to be populated through the ‘customerdata.json’ file. All of this is done by writing the following code:

```
Table table = dynamoDB.getTable("CustomerData");

JsonParser parser = new JsonFactory().createParser(new File("customerdata.json"));
```

Now, the data from JSON is received through tree after the createParser() method has parsed the required JSON file. Once a JSON file is parsed, the data in it gets broken down into different components and forms a tree. The tree comprises multiple nodes, and each node corresponds to an item in the JSON file. The following code snippet shows how data is received in the form of a tree.

```
JsonNode rootNode = new ObjectMapper().readTree(parser);
Iterator<JsonNode> iter = rootNode.iterator();

ObjectNode currentNode;
```

Next, to read through the nodes, a while loop is used, as shown in the code snippet below:

```
while (iter.hasNext()) {
    currentNode = (ObjectNode) iter.next();

    String customerId = currentNode.path("CustomerID").asText();
    String orderId = currentNode.path("OrderID").asText();
```

The while loop will iterate through the tree using the hasNext() method. This method will check whether there is any component present in the tree or not. As long as a component is present, the hasNext() method will transfer that particular component to its relevant object node and move on to the next component. This process will keep repeating itself until there is no component present to read further, and then, the while loop will break causing the iteration to stop.

All the components in the tree will be mapped as string type values. This is done using the asText() method. Once the components are mapped, the items will be added to their designated table using the putItem() method.

To read more about JSON, click here.

Adding a New Item to a Table

A new item can be added to a table anytime by simply writing a Java program. A user can add one or more items in the table by following this process:

To add an item to a table, a user must provide the following details:

1. The name of the table in which the required item needs to be added. This is done using the following statement:

   ```
   Table table = dynamoDB.getTable("CustomerData");
   ```

2. The primary key and sort key (mandatory only if the table also has the sort key) values of the required item. This is done using the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";
```

3. If an item consists of maps or a list of key-value pairs, then all the key-value pairs of that item are specified in the following manner:

```
final Map<String, Object> infoMap = new HashMap<String, Object>();
infoMap.put("city", "Pune.");
infoMap.put("Region", "West");
infoMap.put("State Code", "20");
```

Once all the required details of the item are provided, the item can be added using the putItem() method as shown below:

```
try {
    System.out.println("Adding a new item...");
    PutItemOutcome outcome = table
       .putItem(new Item().withPrimaryKey("CustomerID", customerId, "OrderID", orderId).withMap("info",
infoMap));

       System.out.println("PutItem succeeded:\n" + outcome.getPutItemResult());

      }
```

Reading a Table Item

To read an item of a table, a user must provide the following details:

1. The name of the table in which that item exists. This is done by writing the following statement:

```
Table table = dynamoDB.getTable("CustomerData");
```

2. The primary key and sort key (mandatory only if the table also has the sort key) values of that item. This can be done by writing the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";

GetItemSpec spec = new GetItemSpec().withPrimaryKey("CustomerID", customerId, "OrderID", orderId);
```

3. Finally, the required item can be retrieved using the getItem() method as shown below:

```
try {
```

```
System.out.println("Attempting to read the item...");
Item outcome = table.getItem(spec);
System.out.println("GetItem succeeded: " + outcome);

}
```

Updating a Table Item

To update an existing item in a table, a user must specify the following details:

1. The name of the table in which hat item exists, as shown below:

```
Table table = dynamoDB.getTable("CustomerData");
```

2. The primary key and sort key (mandatory only if the table also has the sort key) values of the item, which need to be updated. This is done by writing the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";
```

3. The values that needs to be updated in the item. In this example, a new key 'hits' needs to be added in the 'info' map, whose value is a number, '1'. The following code snippet tells how it is done:

```
UpdateItemSpec updateItemSpec = new UpdateItemSpec().withPrimaryKey("CustomerID", customerId,
        "OrderID", orderId)

                .withUpdateExpression("set info.hits = :h")
    .withValueMap(new ValueMap().withNumber(":h", 1))
    .withReturnValues(ReturnValue.UPDATED_NEW);

      try {
      System.out.println("Updating the item...");
      UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
      System.out.println("UpdateItem succeeded:\n" + outcome.getItem().toJSONPretty());

}
```

The key that needs to be added is mentioned in the withUpdateExpression method, and its corresponding value is passed through the withValueMap() constructor.

Finally, the item is updated using the updateItem() method.

Deleting an Item from a Table

A table can be deleted by writing a Java program.

To delete an item from a table, a user must provide the following details:

1. The name of the table in which that item exists. For example,

```
Table table = dynamoDB.getTable("CustomerData");
```

2. The primary key and sort key (mandatory only if the table also has the sort key) values of the item that need to be deleted from a table. This is done by writing the following statement:

```
String customerId = "IN-80134";
String orderId = "CA-2018-345678";

DeleteItemSpec deleteItemSpec = new DeleteItemSpec()
    .withPrimaryKey(new PrimaryKey("CustomerID", "IN-80134", "OrderID", "CA-2018-345678"));
```

The item gets deleted using the deleteItem() method, as shown below:

```
    try {
        System.out.println("Attempting to delete an item...");
        table.deleteItem(deleteItemSpec);
        System.out.println("DeleteItem succeeded");
    }
```

Retrieving All Table Items using the Scan Operation

The scan operation is used to retrieve all the items of a table.

To retrieve all the items of a table, a user must provide the name of the table, all the items of which need to be retrieved. After that, the scan() method needs to be used for retrieving all the items of the table, as shown below:

```
    Table table = dynamoDB.getTable("CustomerData");


    try {
        ItemCollection<ScanOutcome> items = table.scan();

        Iterator<Item> iter = items.iterator();
        while (iter.hasNext()) {
            Item item = iter.next();
```

```
            System.out.println(item.toString());
        }

    }
```

Retrieving All Table Items using the Query Operation

The query operation is used to retrieve some specific, not all, items of the table.

To query a DynamoDB table —

1. A user must specify the name of the table, the items of which need to be retrieved using the query operation.
2. Then, the user needs to specify the attribute value, which will act as a filter to produce the item(s) that correspond to this value. Here's the code snippet:

```
HashMap<String, String> nameMap = new HashMap<String, String>();
nameMap.put("#cu", "CustomerID");
```

In the snippet above, the name of the attribute that needs to be referred to must be specified to make a query successful. In this example, the attribute name is 'CustomerID', which is also the primary key. The attribute name needs to be passed through the nameMap.put() method. Here, '#cu' acts as an alias for 'CustomerID'. Later in the code, it will be addressed as '#cu'. Here's the relevant code snippet:

```
HashMap<String, Object> valueMap = new HashMap<String, Object>();
valueMap.put(":c", "BH-11710");
```

In the code snippet above, the attribute value that acts as a filter needs to be specified. This means that only the items corresponding to this attribute value will be searched for and other items will be ignored. Here, 'BH-11710' is the attribute value that acts as a filter and needs to pass through the valueMap.put() method. So, all the items corresponding to 'BH-11710' will be queried. Here, ':c' acts as an alias for 'BH-11710'. Later in the code, it will be addressed as ':c'.

In a nutshell, the two HashMap statements above mean that in the 'CustomerData' table, a user is looking for all the items that correspond only to 'BH-11710'.

3. Next, the attribute value needs to be assigned to the attribute name using the following statement:

```
QuerySpec querySpec = new QuerySpec().withKeyConditionExpression("#cu =
        :c").withNameMap(nameMap)
.withValueMap(valueMap);
```

Here, '**#cu = :c**' means that the attribute name 'CustomerID' is assigned to the value 'BH-11710'.

4. After this, an item collection variable needs to be declared because the query may fetch either one item or a collection of items. When you declare the variable, set its initial value to null, as follows:

```
ItemCollection<QueryOutcome> items = null;
```

In this example, 'items' is the ItemCollection variable, which will store the scanned values once the query operation is executed.

5. Next, you need to loop through the table to scan all the items corresponding to the value ':c':

```
try {
   System.out.println("Orders by CustomerID = BH-11710");
   items = table.query(querySpec);
```

In the statements above, the query() method is used to initialise the scan operation on the 'CustomerData' table. The instance 'querySpec' specifies how to filter the items.

```
        iterator = items.iterator();
    while (iterator.hasNext()) {
          item = iterator.next();
          System.out.println(item.getString("CustomerID") + ": " + item.getString("OrderID"));
       }

    }
```

The while loop then reads each item and adds it corresponding to ':c' in the items' list. The output will be received in the form of strings because the getString() method is used.

Initial Partition Allocation

partition, which is an allocation of storage for a table and is backed by **solid-state drives (SSDs).**

The two primary factors affecting the number of initial partitions in a DynamoDB table are —

- The provisioned throughput capacities
- The size of the data stored in the table

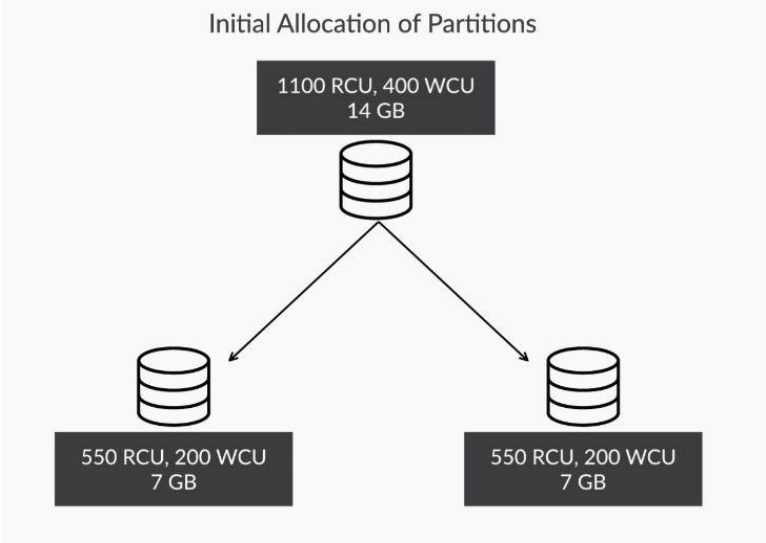
The provisioned throughput capacities, i.e. RCUs and WCUs, determine the number of the initial partitions that are created in a DynamoDB table. Further, DynamoDB divides the throughput capacities evenly among all the available partitions. Each partition can support a maximum of 3000 RCUs and 1000 WCUs. The formula needed to estimate the number of partitions is as follows:

(Desired RCUs/3000) + (Desired WCUs/1000) = Initial partitions (value rounded off to the next integer)

If a table is created with 1100 RCUs and 400 WCUs, then the number of initial partitions created = 1100/3000 + 400/1000 = 0.366 + 0.4 = 0.766. Rounding off this value to the next integer gives a value of 1. Hence, DynamoDB will create 1 partition based on the initial throughput values.

In DynamoDB, each partition can support a maximum data size of 10 GB. The total data is equally distributed among all the initial partitions created by DynamoDB.

If ever size of the data in one partition exceeds 10 GB, DynamoDB will split that partition into two partitions and will divide the data as well as the throughput values evenly across both the partitions, as shown in the image below.



The initial number of allocation of partitions in DynamoDB is the maximum of partitions created based on the throughput requirements and the partitions created based on the size of the data.

For example, if based on throughput values 4 partitions are required and based on size of the data 3 partitions are required then DynamoDB will create 4 partitions.

Secondary Indexing

A secondary index is a data structure created from a subset of attributes from a database and contains an alternate primary key, which can be used to run queries or scan operations. A secondary index helps you access specific data attributes easily, saving both time and money.

DynamoDB offers two types of secondary indexing, which are as follows:

- Global secondary indexing

- Local secondary indexing

Secondary indexing is performed on any created database table, which is also referred to as the base table. It is important to realise here that **though each secondary index is affiliated to only one base table, a single base table can have any number of secondary indexes.**

Table 5: Base Table

Player ID	Player Name	Match Series	Year	Total Score	Strike Rate
101	Virat Kohli	India vs South Africa	2016	450	96
101	Virat Kohli	India vs Australia	2015	360	92
101	Virat Kohli	India vs Sri Lanka	2016	270	92
102	Rohit Sharma	India vs South Africa	2015	287	106
102	Rohit Sharma	India vs Australia	2016	345	92
103	A. Rahane	India vs South Africa	2016	289	101
103	A. Rahane	India vs Australia	2015	328	86
103	A. Rahane	India vs Sri Lanka	2016	221	89

Global Secondary Index: In global secondary indexing, both the partition and sort keys can be different from those in the base table. In other words, the schema of a global secondary index can be distinct from that of the base table. DynamoDB allows you the flexibility of creating up to **five global secondary indexes.**

The steps needed to create a global secondary index are —

- Define **alternate partition and sort keys**. For the base table shown before, define **Match Series** as an alternate partition key and **Total Score** as an alternate sort key.

- Create a global secondary index. The **primary key of a base table** also gets imported to the new index being created. In this example, primary key, **Player ID**, will also get imported to the global secondary index.

- Define the attributes that need to be copied or imported from the base table to the index being created. DynamoDB then copies the defined attributes into the global secondary index. For this example, set **Player Name** as an attribute that needs to be imported along with the alternate composite key from the base table.

Table 6: Global Secondary Index

Base Table Partition Key	Alternate Partition Key	Alternate Sort Key	Imported Attribute
Player ID	Match Series	Total Score	Player Name
101	India VS South Africa	450	Virat Kohli
102	India VS South Africa	287	Rohit Sharma
103	India VS South Africa	289	A. Rahane
101	India VS Australia	360	Virat Kohli
102	India VS Australia	345	Rohit Sharma
103	India VS Australia	328	A. Rahane
101	India VS Sri Lanka	270	Virat Kohli
103	India VS Sri Lanka	221	A. Rahane

Local Secondary Index: A local secondary index has the **same partition key** as the base table, but its sort key and attributes can be different from those of the base table. DynamoDB allows you the flexibility of creating up to **five local secondary indexes.**

The steps needed to create a local secondary index are —

- Create a local secondary index that you need to keep your partition key the same as in your base table, and define an alternate sort key. For this example, define **Player ID** as the partition key and **Total Score** as an alternate sort key.

- While creating a local secondary index, the **sort key value** of the base table is always projected on to the local secondary index, but it is not part of the index key.

- After defining an alternate sort key, you need to define the attributes that need to be copied or imported from the base table to the index being created. DynamoDB then copies the defined attributes to the global secondary index. If you can recall, in our video, we defined **Player Name** as an attribute, which got imported along with the partition key and alternate sort key from the base table.

Table 7: Local Secondary Index

Base Table Partition Key	Base Table Sort Key	Alternate Sort Key	Imported Attribute
Player ID	Match Series	Total Score	Player Name
101	India VS South Africa	450	Virat Kohli
102	India VS South Africa	287	Rohit Sharma
103	India VS South Africa	289	A. Rahane
101	India VS Australia	360	Virat Kohli
102	India VS Australia	345	Rohit Sharma
103	India VS Australia	328	A. Rahane
101	India VS Sri Lanka	270	Virat Kohli
103	India VS Sri Lanka	221	A. Rahane

The differences between global secondary index and local secondary index are as follows:

Global Secondary Index	**Local Secondary Index**
1. A global secondary index can have different partitions and sort keys from those of the parent table or base table.	A local secondary index must have the same partition key as that of the base table, but it can have a different sort key from that of the base table.
2. The index partition key or the sort key (if present) can be an attribute from the base table, of the type string, number, or binary.	The partition key of the index is the same attribute as that of the base table. The sort key can be any base table attribute of the type string, number, or binary.

3. A global secondary index lets you query the entire table, across all partitions.	A local secondary index lets you query a single partition, as specified by the partition key value in the query.
4. A global secondary index can be added to an existing table.	A local secondary index always needs to be defined when creating the table.
5. A global secondary index has its own provisioned read-and-write throughput values.	A local secondary index has the same provisioned throughput values as those of the base table.
6. For each partition key value in a global secondary index, there is no size restriction.	For each partition key value in a local secondary index, the total size of all the indexed items should be 10 GB or less.

Lecture Notes

Functional Programming

This session introduced you to functional programming and explained how it is different from the imperative programming languages like Java, C++, etc. Functional programming is slowly becoming the preferred style for developing programs or applications for big data processing. You also learnt the exact reasons why functional programming is preferred over imperative programming for big data processing.

Understanding Imperative Programming

Imperative Programming is the style of programming which solves a problem by assigning values to variables. This means Imperative Programming extensively makes use of mutable variables or objects i.e. variables or objects whose state can change anytime during the program's execution lifecycle. The features of Imperative programming model are closely related to machine architecture based on the Von Neumann model of stored program computer. The imperative model has taken inspiration from machine language or machine code. The machine language is the language which the CPU understands.

You were introduced to imperative programming with the help of below mentioned example:

```
int num = 5;

long fact = 1;

for (int i = 1; i <= num; i++)
      {

         fact = fact * i;

      }
```

The key observations from the above code are mentioned below:

- The variables used in the code are **num, fact** and **i.**
- The value of variable 'num' remains constant throughout the execution of the program whereas the values of variables 'fact' and 'i' do not remain constant and change in each iteration of 'for' loop i.e. the state of the variables fact and i is not constant.

This style of programming in which the primary method of computation is by assigning values to variables is known as **imperative programming**.

The features of Imperative programming model are closely related to machine architecture based on Von Neumann model. The imperative model has taken inspiration from machine language or machine code. Machine language is the language which the CPU understands. All high-level languages are first converted into machine code and then processed by CPU. In the context of machine languages, the contents of memory, registers, etc. determine the state of a computation or the program. Registers are data holders designed to store specific information like instructions, the address of the next instruction to be executed, a value of an operand etc. CPU executes these machine codes by adding, say two numbers, updating the value of a variable or deleting the contents of memory or registers. This means, the state of the program i.e. contents of a memory location, final value in a register etc. are bound to change whenever a machine code is executed.

The various high-level imperative languages are broadly divided into three categories:

- **Structured**: These languages provide constructs like if-else, for loop, while/do while, etc to ensure better clarity, readability and maintainability of the code. Eg. Algol 60
- **Procedural**: These languages introduced procedure or subroutine, which is a sequence of instructions which are executed to accomplish a specific task. A procedure can be called anywhere from the program. Eg. Pascal and C.

- **Object-oriented**: The data and its associated methods are enclosed in a single entity known as an object. Eg. C++ and Java.

Impure Functions

Imperative programming makes use of impure functions. Features of impure functions are:

- Apart from the parameters passed as arguments can rely on the external state for input.
- Can even modify the external state as output.

When an external state is used as input this external state is termed as a **side cause** whereas when an external state is modified as output then this modification is termed as a **side effect.**

```
public List getMovies(String genre) {

  if (genre.equals("Hollywood"))
      return getEnglishMoviesForDate(new Date());

  else if (genre.equals("Bollywood"))
      return getHindiMoviesForDate(new Date());
}
```

In the function `getMovies()`, the actual input to the function is "genre". Internally, the function makes use of another input which is "new Date()" i.e. the current date. Therefore, the output of the function will vary based on the day on which this method is executed. So, new Date() is a **side cause** of this function because this is not present in the parameter list of the function.

Some examples of side effects are:

```
private static int flag = 1;

public int mulValue(int a, int b){

      if (flag == 1)
          flag = 0;
      else
          flag = 1;
      return a * b;
}
```

The function mulValue takes two parameters a and b, multiplies both of them and returns the product. Apart from determining the product, the function also modified a static variable flag. This modification is termed as a side effect because it has done changes to a variable which was not included in its parameter list.

Pure Functions

The functions which do not rely on a side cause and do not produce a side effect are known as pure functions. An example for pure functions is mentioned below:

```
public List getMovies(String genre, Date movieDate) {

  if (genre.equals("Hollywood"))
      return getEnglishMoviesForDate(movieDate);

  else if (genre.equals("Bollywood"))
      return getHindiMoviesForDate(movieDate);
}
```

After the addition of the parameter movieDate, the function getMovies() is converted to a pure function. Now the function is completely free from any external input or side causes. Hence, this function is pure as it is completely dependent on its input parameters for computing the result and does not have side effects.

The features of pure functions are:

- Pure functions are inspired by mathematical functions like sine, cosine, square, etc.
- Pure functions are deterministic in nature. A function is deterministic if it returns the same output for a given input. If we have a square function, then the function will always return 16 whenever it is executed with input 4.
- Pure functions are free from side effects and side causes.
- Pure functions are easy to test and maintain because pure functions are deterministic in nature. Pure functions can be tested in isolation because of their heavy reliance on input parameters and independence from the external world.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Functional Programming

Functional programming is a style of programming which extensively makes use of pure functions only.

Some characteristics of functional programming are:

- Encourages the use of immutable objects and variables for avoiding state change
- Instead of iterations, functional programming languages use recursions to solve an iterative problem

For e.g. factorial of a number can be computed using a for loop i.e.

```
for (int i = 1; i <= num; i++)
{
      fact = fact * i;
}
```

The same task can be performed using recursion in functional programming:

```
fact(int num){
      if (num == 1)
            return 1;
      else
            return (num * fact(num-1))
}
```

- Functional programming languages are not memory efficient because a change of state is not allowed.
- Functions are treated as first-class citizens in functional programming. This means functions like variables can be passed as parameters in other functions. Functions can also be assigned to variables.

Example of a function which can take another function as a parameter is the map function or reduce function which we had discussed in this course in the earlier modules. The map function will take the list of data and the function as input.

Some of the reasons why functional programming languages are used for developing jobs for processing big data are:

- As functional programming languages extensively make use of deterministic pure functions, the input values can be distributed across multiple computing devices and the pure functions are executed independently on these machines.
- Supports memoisation because pure functions are referentially transparent which means for a given set of input values the function will always return a fixed output. So, the function is only computed once, and the output is stored in a cache. Whenever this value is required, without recomputing the process can directly lookup from the cache memory.
- Functional programming languages establish a functional dependency between the input and the output values. So, if a certain output is not required then its evaluation can be delayed. This delay in evaluation is known as lazy evaluation.

Lecture Notes

HBase

In this module, you were introduced to HBase — a NoSQL database component of the Hadoop ecosystem and the features provided by it that differentiate it from Hadoop and other relational or NoSQL datastores. You started with understanding the limitations of SQL and the HDFS, and then were introduced to NoSQL databases. After looking at the reasons for their increasing popularity, you learnt about the CAP theorem and its consequences, specifically, the tradeoffs between consistency, availability, and partition tolerance in the design of network sharing-enabled data systems. Finally, you looked at how data is stored in HBase and understood its data model.

NoSQL Databases

Databases like RDBMS, which support SQL, can store data in tables divided into rows and columns. Here, each row represents a data record, which comprises a collection of columns or attributes. A fixed schema governs the tables, which means that a data type is associated with each column. In the case of data type violations, while inserting data into a table, the insert operation fails, and there's no flexibility in the data model to accommodate such discrepancies in data.

A record is added successfully to an SQL table only when the types of all the individual data elements in the record strictly adhere to the destination table's schema. Because of such strict schema-oriented regulations, SQL databases are used for storing and processing only structured data.

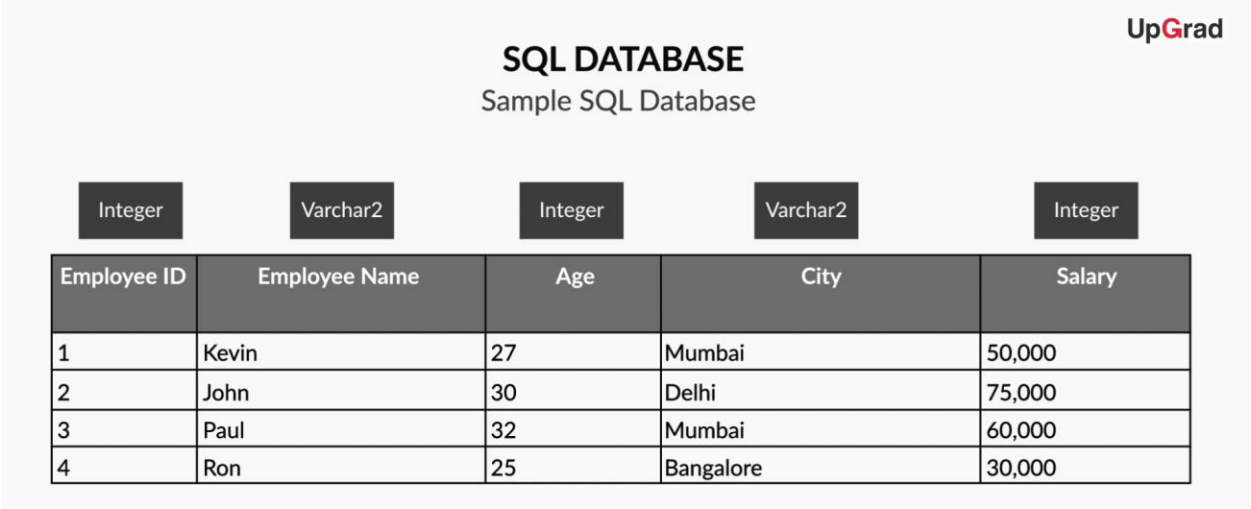


Figure 1: SQL Database

With technological advancements that brought about the ease in the availability of internet at affordable prices, increased usage of IoT devices, an abundance of social networking websites, etc., the rate of generation of digital data grew at an alarming pace in the last couple of decades. Hence, traditional systems were not scalable enough to accommodate such massive volumes of data. Moreover, the digital data was not structured anymore, and organisations were not interested in discarding this data because, nowadays, non-structured data comprises almost 80% of the total generated data. And 80% is a significant proportion of the total data.

So, to store and process this humongous non-structured data, organisations started to migrate from traditional SQL-supported systems to other data-processing tools, which were scalable and fault-tolerant.

When organisations were trying to find out a way to store and process big data, Hadoop came to their rescue. Hadoop uses a file system called the Hadoop distributed file system, or HDFS, as its storage layer. This file system stores data in a distributed manner across a cluster of commodity machines. The HDFS is a robust file system, which can store structured, unstructured, and semi-structured data besides being horizontally scalable.

Though Hadoop overcame some of the challenges faced by SQL, it introduced new challenges. Hadoop MapReduce (used for processing the data stored in the HDFS) is well-suited for batch processing, where the whole data is accessed sequentially. But the drawback of using MapReduce for processing big data is that it is not suited for all use cases, such as performing random lookups on data. Hence, apart from Hadoop, a solution was needed for these use cases. These limitations of Hadoop led to the inception of NoSQL datastores.

Some reasons for the increasing popularity of NoSQL databases are —

- NoSQL datastores are efficient in storing and handling big data. Based on the targeted use cases, every NoSQL database has its own data model for storing data.

- NoSQL datastores provide scalability, i.e. in case of space crunch, extra space can easily be created by just adding additional nodes to the cluster.

- NoSQL datastores are flexible and do not restrict themselves to a fixed schema. Hence, they can dynamically adapt to the changes in the schema of the data.

CAP Theorem

The CAP theorem relates to the distributed systems that store data. The CAP theorem states that it is impossible for a distributed datastore to simultaneously provide more than two out of its three guarantees — consistency, availability, and partition tolerance. Consistency guarantees that every node in the distributed system returns the same, most successful and recent write. Availability ensures that every request receives a response, without the guarantee that it contains the most recent write. Partition tolerance confirms that the system continues to function upholding its guarantees in spite of network partitions.

Some of the known systems and guarantees provided by them are mentioned below:

- **RDBMS Systems (support SQL):** Consistency and availability.

- **NoSQL Systems:** They store data in a distributed manner across a cluster of interconnected machines and provide network partitioning. Hence, there are two variants of NoSQL databases, and they come with different sets of guarantees:

    1. Consistency and partition tolerance (e.g. HBase, MongoDB)
    2. Availability and partition tolerance (e.g. Cassandra, DynamoDB)

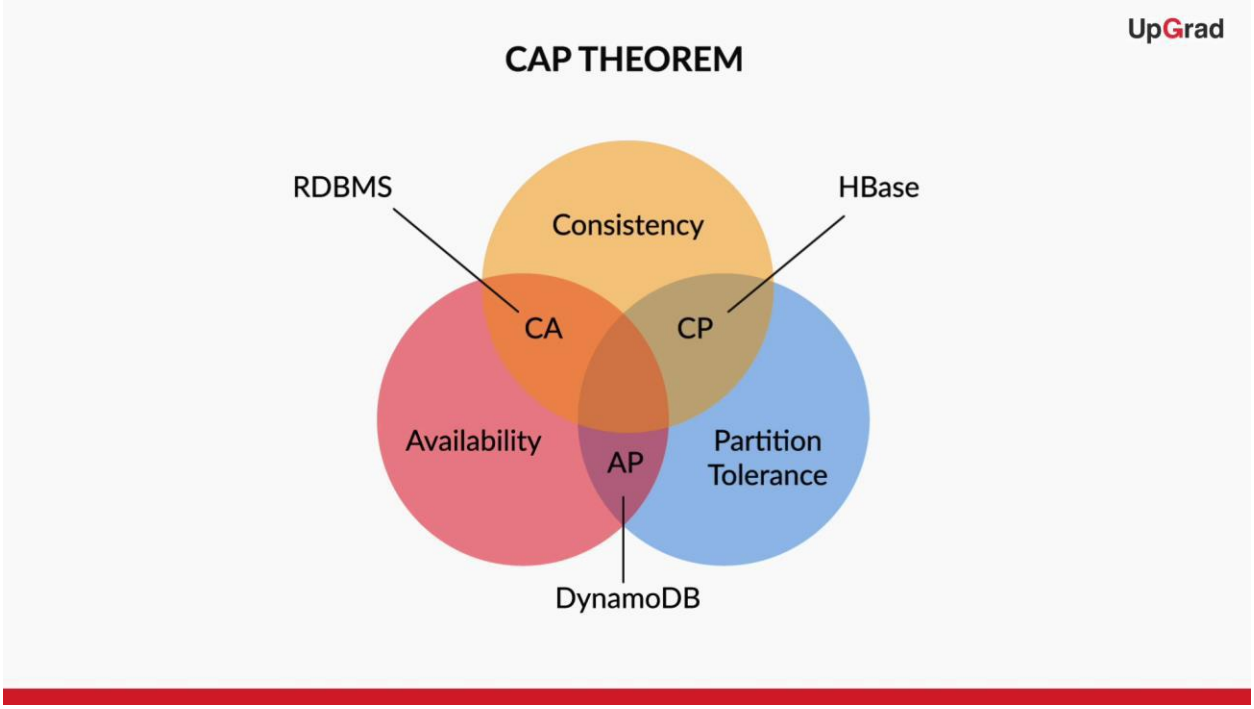


Figure 2: The CAP Theorem

Depending upon the type of application, while designing network sharing-enabled data systems, system designers should make the tradeoff between consistency, availability, and partition tolerance wisely.

If one chooses consistency over availability, the system will report an error or a timeout if particular information is not the latest due to network partitioning. Consider the example of using a chat application, where you have sent an instant message to your friend. If, during this process, there is a network partition then, ideally, the message will not get delivered, and there will be a timeout. Once the system is up, the exact message you sent will be delivered to your friend's inbox. So, in a messaging application, consistency is given preference over availability. By the same token, just to ensure instantaneous delivery of messages, the system should not deliver a garbage message.

When one chooses availability over consistency, the system processes the query and returns the newest version of information available, even if the data is not the latest due to network partitioning. Consider another example of using a travel portal like MakeMyTrip, where you can check the availability of hotel rooms. Due to network partitioning, if the portal is unable to fetch the latest prices, it would be perfectly fine because the customer may not be concerned about prices. He/she may be concerned about other aspects such as the location of the hotel,

amenities, etc. So, in such scenarios, using a highly available datastore is a must because travel portals and e-commerce websites cannot afford website timeout for showing consistent results.

HBase and Its Features

HBase is a distributed datastore built on top of the HDFS and can leverage all the benefits provided by Hadoop or the HDFS. It has the ability to allow a user to query for individual records as well as derive aggregate analytic reports across a massive amount of data. HBase was first released as a code for an open-source BigTable implementation.

Some of the prominent features of HBase that distinguishes it from Hadoop and other relational or NoSQL datastores are —

- It stores data internally as key-value pairs where the RowKey is the key, and the rest of the data is its value
- Its records are sorted by its row keys
- its columns do not have any specific data type, and all the data in an HBase database is stored in the form of bytes
- It does not follow a strict schema (schema-less and dynamic architecture) which means that any number of columns can be added dynamically

Data Model of HBase

Like SQL, HBase also stores data in a tabular format with some modifications and the data model of HBase can be summarised as follows:

- In HBase, data is stored in tables, which are nothing but a collection of rows. In an HBase table, a row is a collection of column families. A column family is a collection of related columns known as column qualifiers, and there can be any number of columns in a single column family.

- Every entry in an HBase table is identified and indexed by a RowKey, and for every RowKey, an unlimited number of columns can be stored. This feature ensures that the schema of an HBase table is flexible and the table can scale linearly.

- Each column can have a configurable number of versions, and there is a provision for selecting data from a particular version. In HBase, each version is identified by its timestamp, and each column can have one or more versions.

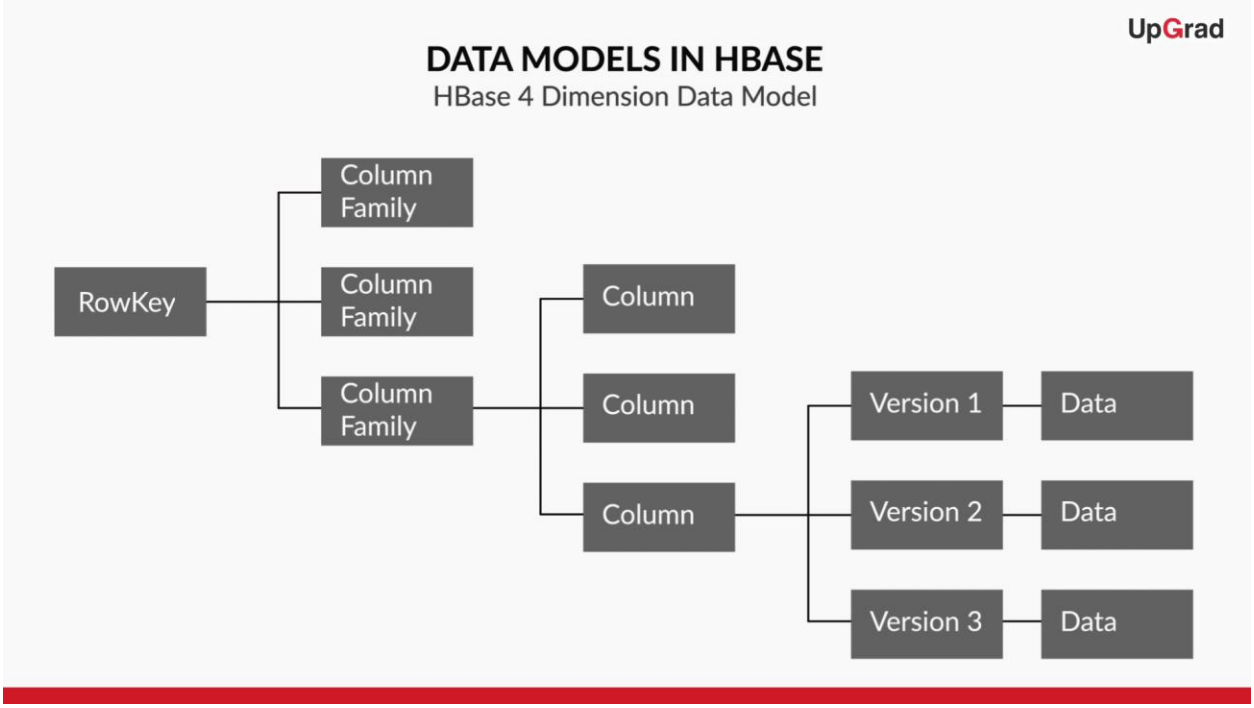


Figure 3: Data models in HBase

To access an individual piece of data, one needs to know its row key, column family, column qualifier, and version number, which makes it a four-dimensional model.

Shell Commands in HBase

HDFS, along with MapReduce, follows the WORM (write once and read many times) paradigm. In other words, data in HDFS is written once, but it can be read an unlimited number of times. Also, there is no provision for updating an existing dataset or record in HDFS. Even though HBase uses the HDFS to store data, it supports update operations by maintaining multiple versions of same the data points.

Shell commands used for performing CRUD (Create, Read, Update and Deletion) operations in HBase are —

- **List:** This command is used to check all the tables present in your HBase instance.

  Syntax: **hbase> list**

- **Create:** To create a table in HBase, you can use this command. To create a table, you must name the table and define its schema. As part of the schema, you are required to

specify the column families. However, columns are defined later when inserting records into the HBase table.

Syntax: **hbase> Create '<table_name>', '<column_family_name>'**

Example: Create a table named 'Students' with three column families — Personal Details, Contact Details, and Marks.

Command: Create 'Students', 'Personal Details', 'Contact Details', 'Marks'

- **Scan:** This command is used to view the contents of an existing table. The optional parameters in its syntax include TIMERANGE, FILTER, TIMESTAMP, LIMIT, MAXLENGTH, COLUMNS, CACHE, STARTROW, and STOPROW.

  Syntax: **hbase> Scan '<table_name>' {Optional parameters}**

- **Put:** This command is used to add or update a cell value in the mentioned table. Each cell in a table can be identified by a row ID and a column name.

  Syntax: **hbase> put '<table_name>', '<row_key>', '<column_value>', '<value>'**

  Example: Insert a 'Name' record (student name 'Sandeep') into the 'Students' table in the row 'students1' and the column 'Personal Details'. After the insertion, update the earlier record by adding the 'email ID' of Student1.

  Command: Put 'Students', 'Student1', 'Personal Details:Name', 'Sandeep'

  Update command:
  Put 'Students', 'Student1', 'Personal Details:Email', 'Sandeep@pqr.com'

- **get:** This command is used to fetch data from an HBase table. There are various ways to make use of the get command.

  Syntax: **hbase> get '<table_name>', '<row_key>', {'<Additional Parameters>'}**

  Example: Fetch data from the 'Students' table in the three ways mentioned below:

    1. Fetch the entire 'Student1' row data

2. Fetch data from the column family named ‘Personal Details’

3. Fetch data from the column named ‘Name’”

Commands:

1. get ‘Students’, ‘Student1’

2. get ‘Students’, ‘Student1’, {COLUMN => ‘Personal Details’}

3. get ‘Students’, ‘Student1’, {COLUMN => ‘Personal Details:Name’}

- **delete:** To delete a cell in an HBase table, you can use this command.

  Syntax: **hbase> delete ‘<table_name>’, ‘<row_key>’, ‘<column_value>’, ‘<value>’**

  Example: Delete the ‘email ID’ updated in the ’Student1’ record.

  Command: delete ‘Students’, ‘Student1’, ‘Personal Details:Email’

Note:

1. If you observe the Put command, you would understand that columns are defined at the time of inserting records into the HBase table (The column ‘Name’ is defined while inserting the name ‘Sandeep’.)
2. ‘<Additional Parameters>’ mentioned in the syntax include TIMERANGE, TIMESTAMP, VERSIONS, and FILTERS

3. The ‘scan’ command helps to verify whether the contents of a table are deleted or not

Additional Shell Commands

Shell commands that are used to modify and view the specifications of an HBase table include:

- **describe:** This command is used to check the schema of an HBase table.

Syntax: **hbase> describe <table_name>**

- **exists:** This command is used to verify whether a given table is present in the HBase storage or not.

  Syntax: **hbase> exists <table_name>**

- **alter:** This command is used to modify the specifications of an HBase table, such as adding column families, updating the version number of the column families, etc. It can also be used to delete the column family by applying the delete method to it.

  Syntax (modifying the number of versions for one column family):
  hbase> alter '<table_name>', Name=>'<column_family_name>', VERSIONS => <new_version_number>

  Syntax (modifying the number of versions for multiple column families):
  hbase> alter '<table_name>', {Name=>'<column_family_name>', VERSIONS => <new_version_number>}, {Name=>'<column_family_name>', VERSIONS => <new_version_number>}

  Syntax (deleting a column family):
  hbase> alter '<table_name>', Name=>'<column_family_name>', METHOD => <'delete'> or hbase> alter '<table_name>', 'delete' => '<column_family_name>'

- **drop:** This command is used to delete a cell in an HBase table. But this operator cannot be applied directly to the table. Instead, the table is first disabled. And then it is dropped.

  Syntax:
  Step 1: **hbase> disable '<table_name>'**
  Step 2: **hbase> drop '<table_name>'**

- **truncate:** This command is used to remove all the data from a table (note that we do not intend to delete the table, just the data that is stored in the table). Internally, this command disables the table, drops it, and recreates it again. But to us, the end result is that the table's data has been removed.

  Syntax: **hbase> truncate '<table_name>'**

HBase stores multiple versions of the same data present in a single cell, and that data stored in HBase tables can be filtered using shell commands.

- **Get data based on timestamps:** The 'get' command can also be used to retrieve past versions of records based on timestamps.

- Syntax: **hbase> get '<table_name>', <row_key>, {COLUMN => '<column_family_name>', TIMESTAMP => value}**

- **Get data based on filter conditions:** In HBase, fetching data based on a filtering condition is achieved by using filters. Here, filters are like Java methods that take two input parameters; these include a logical operator and a comparator. The logical operator specifies the type of the test, i.e. equals, less than, etc. The comparator is the number/value against which you wish to compare your record.

  Some commonly used filter functions are —

  1. **ValueFilter:** A ValueFilter takes a comparison operator and a comparator as its parameters. It compares each value with the comparator using the comparison operator. If the check returns true, then the result is displayed on the console.

     Syntax: **ValueFilter (<compareOp>, '<value_comparator>')**

     Let's learn this through an example of using ValueFilter with the scan command:

     Consider an example HBase table named 'Companies', which is maintained by the placement cell of a college and contains the details of all the companies that visit the college every year. Check whether the company 'UpGrad' exists in that table or not.

     Command: hbase> scan 'Companies', {FILTER => "ValueFilter(=, 'binary:UpGrad')"}

  2. **QualifierFilter:** A QualifierFilter also takes two parameters; they are comparison operator and comparator. Each qualifier name is compared with the comparator using the compare operator, and if the comparison is true, it returns the key-value pairs in that column.

     Syntax: **QualifierFilter (<compareOp>, '<qualifier_comparator>')**

Let’s learn this through an example of using QualifierFilter with the scan command:

Consider an example HBase table named ‘Companies’, which is maintained by the placement cell of a college and contains the details of all the companies that visit the college every year. Find out all the names of the companies present in the table (in the HBase table given, the names of the companies are present in the column named ’Name’).

Command: hbase> scan ‘Companies’, {FILTER => “QualifierFilter(=, ‘substring:Name’)”}

3. **FamilyFilter:** A FamilyFilter is used to fetch key-value pairs for a specified column family.

   Syntax: **FamilyFilter (<compareOp>, ‘<family_comparator>’)**

   Let’s learn this through an example of using FamilyFilter with the scan command:

   Consider an example HBase table named ‘Companies’, which is maintained by the placement cell of a college and contains the details of all the companies that visit the college every year. Fetch the contact details (which include the mobile number, email ID, etc.) of all the companies present in the table (in the HBase table given, the names of the companies are present in the column family named ’Contact Details’).

   Command: hbase> scan ‘<table_name>’, {FILTER => “FamilyFilter(=, substring:Contact Details’)”}

- **count:** This command is used to count the number of rows present in a table.

  Syntax: hbase> count ‘<table_name>’

Lecture Notes

Introduction to Big Data

This session helped you understand what exactly big data is and how it is different from the data that organisations used 15-20 years ago. It is always better to understand the data before performing any action on it, because this information will help you choose the appropriate processing technique, which will eventually lead to the selection of suitable big data processing tools.

In this session, starting from the core concept of data, you moved towards understanding the notion of big data and the features that come with it. Also, you took a look at what the problem with big data is and why it needs to be solved.

After completing this session, you have thoroughly understood the importance of the various nuances of big data.

What is Big Data

In the lectures, we discussed that big data shares the same definition as data, i.e. “some existing information or knowledge is represented or coded in some form suitable for better usage or processing,” with the only difference that it is enormous in size.

It’s not only about the present volume of the data, but also that the size of the dataset may not remain static. Big data has the potential to grow exponentially for an indefinite period. It can increase even to the extent where it can’t be managed or processed using traditional techniques such as RDBMSs. The graph below shows the trend of user growth on LinkedIn:

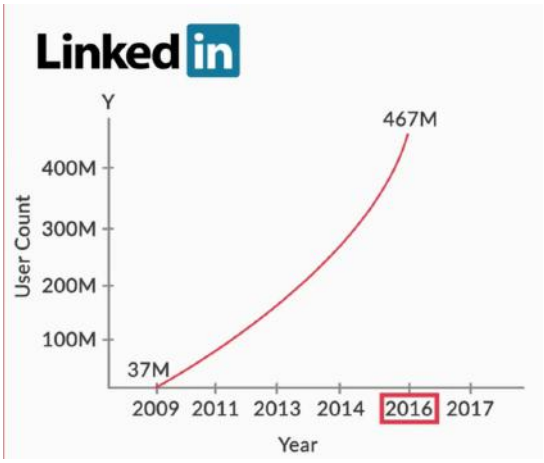


Figure 1: LinkedIn user growth trend

One of the definitions for big data, as given by our industry expert, is:

"If the data has expanded to such an extent that now a single computing system is unable to store and process it, then we can call this data big data."

If you want to quantify big data in terms of size, then you need to consider the maximum possible capacity of an HDD (hard disk drive) available today, which is 16 TB. Using this size as a scale, any data volume in terabytes or petabytes would be considered as big data because it would be difficult for a single system to accommodate a dataset in the range of TBs or PBs.

Big Data: Interesting Facts and Statistics

With the increasing dependence on the internet, every online user activity, such as a Google search, a 'like' on a Facebook post, or sending/receiving of an email, leads to data generation. Refer to Figure 1 to understand the data explosion happening in one internet minute. In an Internet minute on YouTube, 300 hours of video is uploaded, and 1.3 million videos are viewed. In that minute, more than 2 million searches are made on Google, and approximately 350,000 new tweets are tweeted on Twitter. There are 180 million active websites in the world and growing. Thus, you can imagine the cumulative growth of the amount of

data generated every minute.

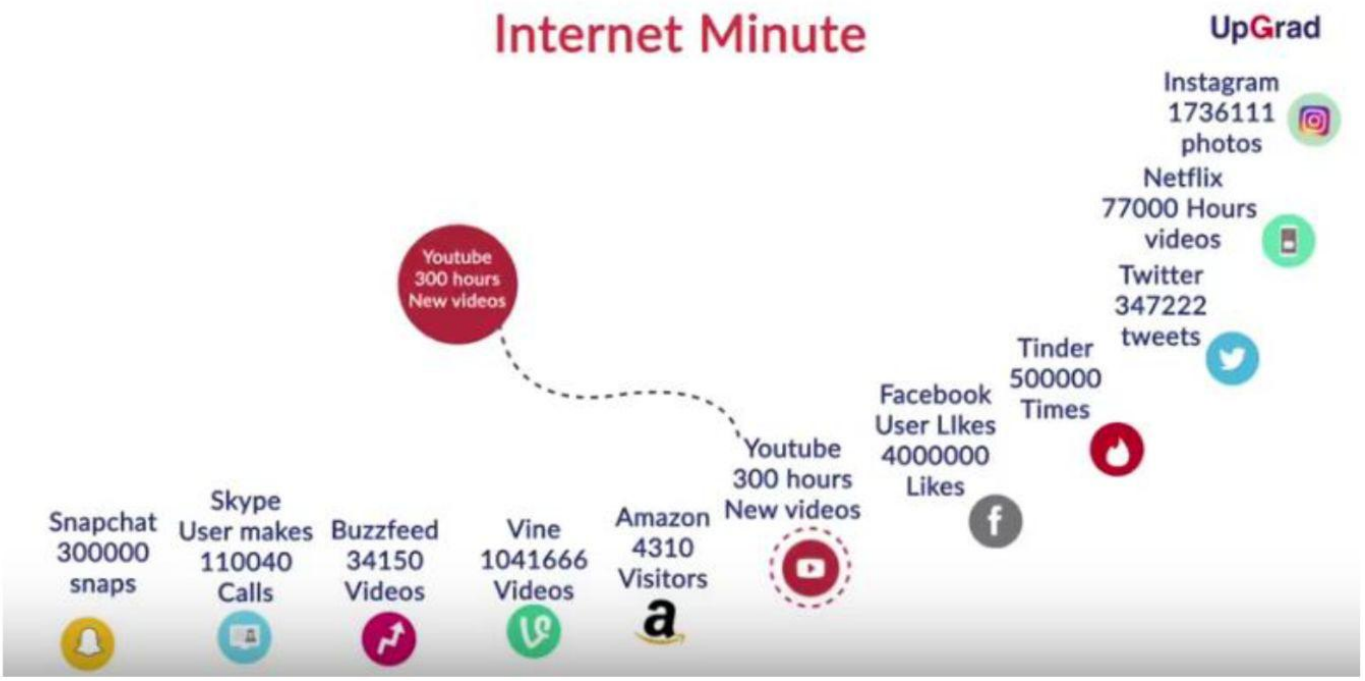


Figure 2: Data explosion in an internet minute

So how big is big data? Does it have to be as big as the data held by Google? Is Facebook's data big enough to be designated as big data? Well, here are a few statistics that'll answer the questions. Refer to figure 2 and 3 to get a fair amount of idea regarding the amount of data Facebook and Google deal with, respectively.

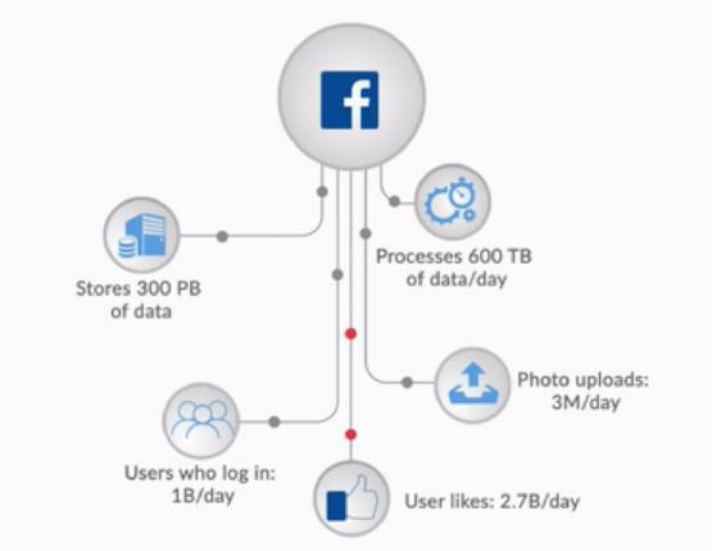


Figure 3: Facebook's Big data facts

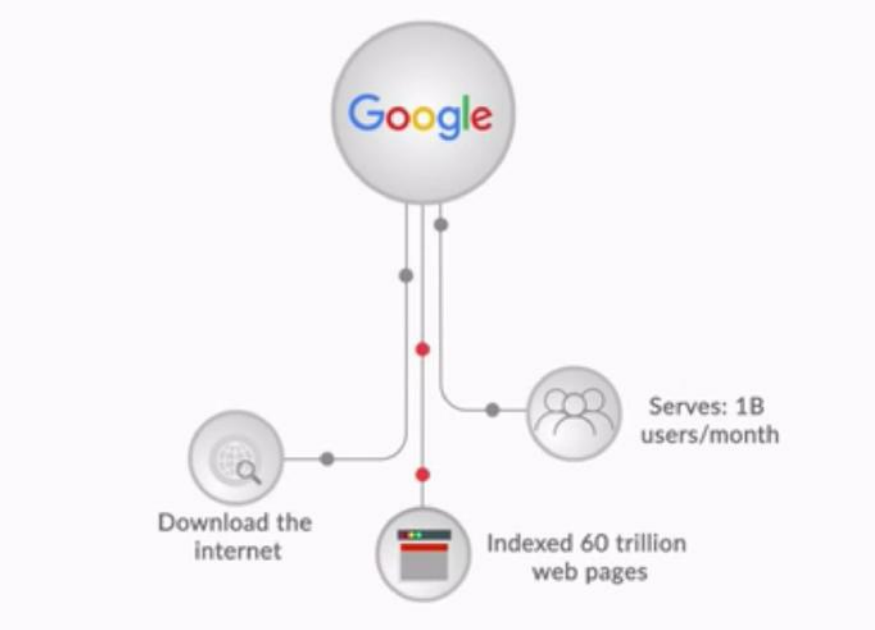


Figure 4: Google's Big data facts

To reiterating what has already been discussed, big data doesn't refer to any specific quantity. Wherever the available infrastructure can't handle the incoming data, it is designated as big data for that set-up. The term is often used when speaking of Petabytes, Exabytes, or even Terabytes of data.

Big Data: Some Basic Industry Application

As starters, just to get a feel of the potential of big data in an industrial set-up, we discussed some of the widely used applications of big data in various industries. They are —

- Application of big data in the healthcare industry
- Application of big data in the retail industry
- Application of big data in the finance industry
- Application of big data in the manufacturing industry

Let's look at each industrial set-up and its big data application in the following sections.

Big Data in the Healthcare Industry:

People visit doctors for consultation. It's a common practice that after detecting a patient's symptoms, the doctor shall suggest some tests. The test reports could either be a paper document, such as a blood test report, or it could be an image, such as a CT scan or an X-ray report. So, all these various test data is stored in a data store, and they are processed and analysed to gain valuable insights.

Figure 5: Big Data in the healthcare industry

Some of the applications of the insights or patterns derived from the analysis of a test report are —

- Real-time monitoring of patients
- Predicting outcomes or upcoming health-related hazards
- Saving cost and energy by minimising hospital visits

Big Data in the Retail Industry:

In the retail industry, the daily sales transactions are recorded and analysed. The rate of data generation is tremendous, and the volume of data is an exorbitant amount as well. This sales data is stored and studied

to discover previously unknown trends and patterns.

Figure 6: Big data in the retail industry

Some of the applications of the insights or patterns derived from the analysis of retail sales data are —

- Understanding customer preferences
- Creating a 360-degree view of a customer profile
- Buying patterns of various products

Big Data in the Finance Industry:
Some advantages of analysing and applying the insights derived from financial data are —

- Detecting and stopping fraudulent transactions
- Designing and modifying predictive models on investment strategies

Big Data in the Manufacturing Industry:
Some of the applications of manufacturing big data analysis are —

- Reading and analysing data from sensors attached to various machine parts
- Proactive maintenance of equipment
- Preventing the loss of machine hours

After learning about the benefits of these industrial applications, we discussed the various aspects of big data. They are —

- **Volume**: The size of the data has to be huge, i.e. in the range of terabytes or even more than that.

- **Rate of change**: The nature of the data has to be dynamic because of the changes in transactions. There could be multiple reasons supporting the changes in transactions, such as a change in the business logic or a change in the requirements.

- **Variety**: Based on the form of data, it can be broadly divided into three categories:
    - **Structured**: The data is stored in a tabular format
    - **Unstructured**: Data that does not have a well-defined structure, e.g. videos and images
    - **Semi-structured**: Data that is partially structured or is a combination of both structured and unstructured data. E.g. emails. Emails are semi-structured because they have a well-defined structure that consists of a sender address, receiver addresses, subject, message body, attachments, etc. But the content mentioned in the subject or the message body is entirely unstructured.

Big Data: Major Sources

You learnt that the sources of big data can be broadly classified into three major categories:

- **People**:

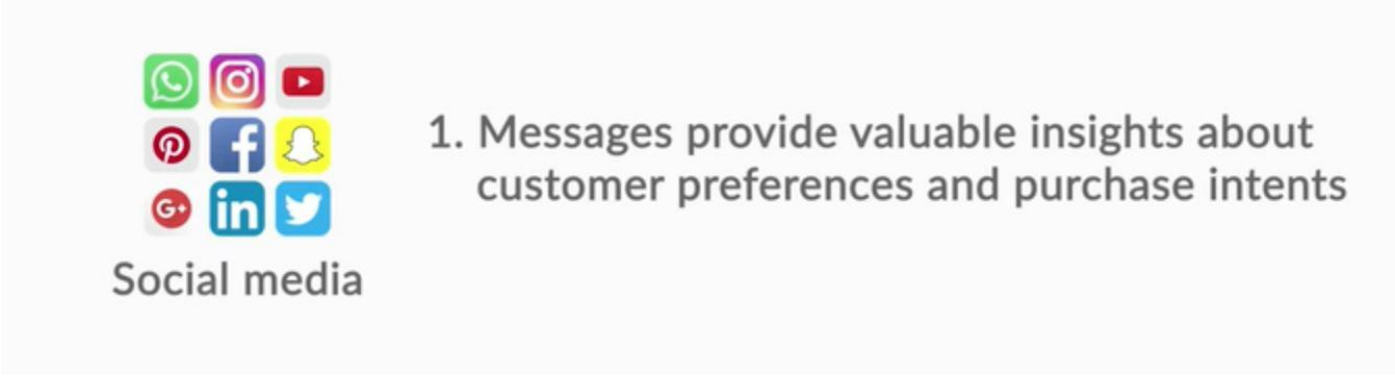


Figure 7: Social media data is an example of data generated by people

Today, people are quite active on the internet through social networking sites such as Facebook, Twitter, and Instagram. On these platforms, they share a lot of information

including what could be a valid opinion regarding a political issue or a post about their recent visit to a hill station. This is considered as data that is 'shared' by people. Even the user ratings for a movie or product can be treated as data generated by people.

- **Machine**: Data that is generated by a machine/computer in a periodic manner or at the occurrence of some event is termed as 'data generated by machines'. Some common examples of data produced by machines are —
    - Data produced by cell phone towers
    - Data produced by RFID tag scanners
    - Data produced by car sensors

Let's look at some examples of data generated by machines through images:

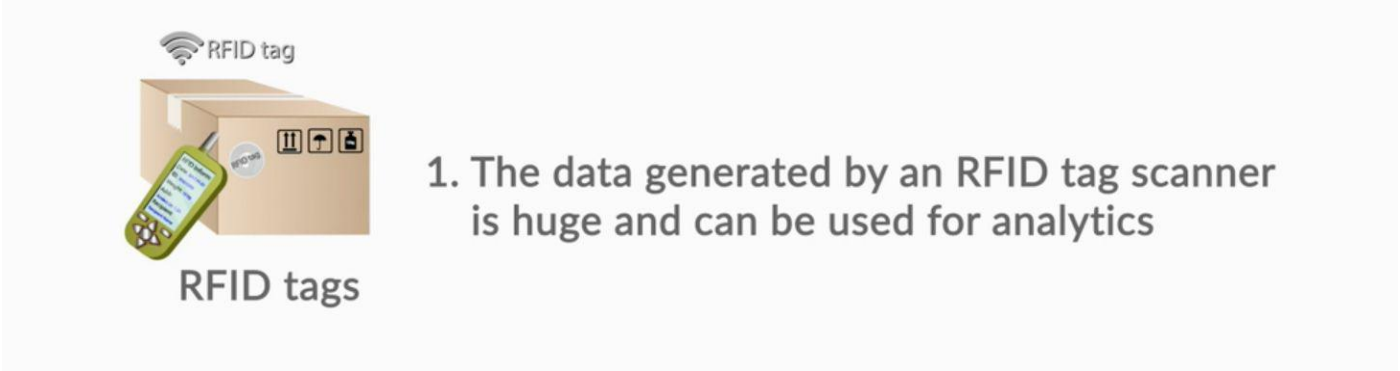


Figure 8: RFID scanner data is an example of data generated by machines

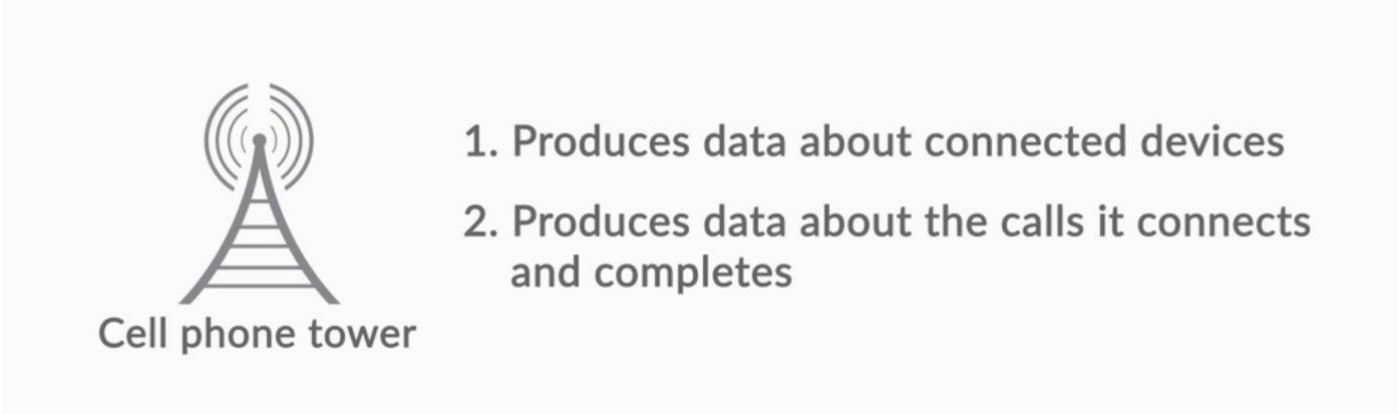


Figure 9: Cell phone tower data is an example of data generated by machines

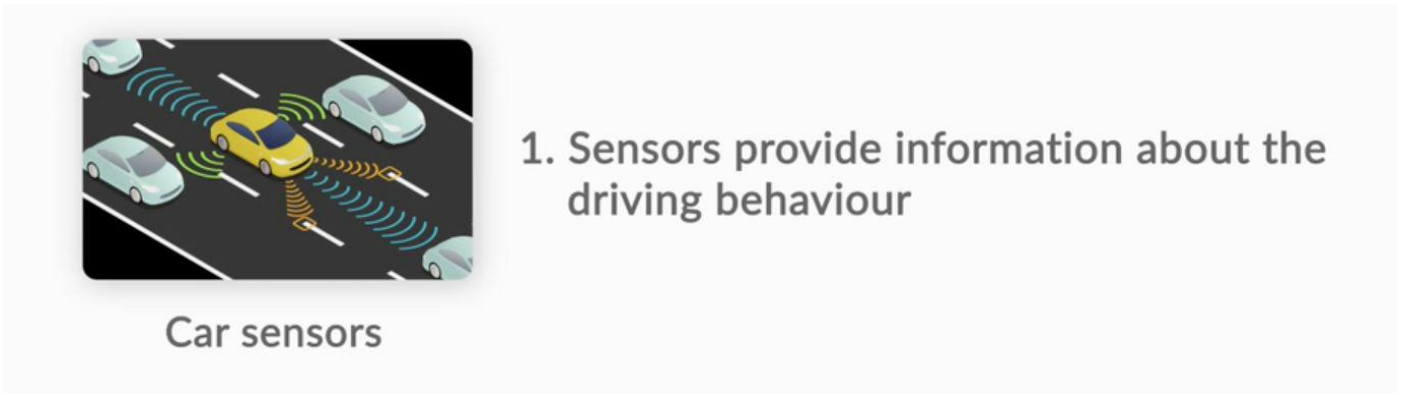


Figure 10: Car sensor data is an example of data generated by machines

- **Organisation**: This refers to the data that is generated by an organisation. This data, in most cases, has a well-defined structure. Some examples of data produced by organisations are internal sales data and customers' demographic information. This kind of data can be integrated with some external data to generate useful insights. Archived organisational data is helpful in performing comparative analyses with the latest data, as well as historical analyses, which are helpful in the prediction of future trends.

Big Data: Major Types

Data is broadly categorised into three categories. They are —

- Structured
- Semi-structured
- Unstructured

The characteristics and examples of the aforementioned data types are mentioned below:

Name	Characteristic	Example
Structured	1. Constitutes 5% of all the information being processed. 2. Has a definite schema or structure	1. Stored in DB tables
Semi-structured	1. Constitutes 5-10% of all the information being processed 2. Cannot be stored in DB tables 3. Metadata provides structure	1. XML 2. JSON 3. Data from social networking websites 4. Email and server logs
Unstructured	1. Constitutes 80% of all the information being processed 2. Does not have a definite schema or structure	**1. Human-generated** a. Social media activities b. Uploaded photos, presentations, documents, videos, etc. **2. Machine-generated** a. Satellite images b. Weather data c. CCTV footage d. RADAR and SONAR data e. Vehicular data

Vs of Big Data

Until this point, you understood the two essential characteristics of big data, i.e. it is vast and diverse. Big data is said to be diverse because it can include structured, semi-structured, and unstructured data. Apart from these two characteristics, big data can also be described with other characteristics. These are represented using the 4 Vs, often denoted as the '4 Vs of big data'. They are —

- **Volume**: This represents the amount of data generated by a company/organisation. The size of big data typically ranges from petabytes to exabytes. For example, remember the amount of data generated in an internet minute, which leads to data explosion. Search engines such as Google, Yahoo, etc. deal with enormous volumes of data too.

- **Velocity**: This indicates the rate at which data is generated/consumed. Social media sites such as Twitter, Facebook, etc. create data from every activity that a user performs, leading to an enormous amount of data generated every minute.

- **Variety**: This represents the different types of data being generated. For example, the various types of data collected from Gmail may be sign-up/registration data, user login data, inbox emails, sent emails, etc.

- **Veracity**: This represents the quality and trustworthiness of data. Previously, veracity was not considered to be a characteristic of big data. But with the increasing analyses on generated data, veracity plays an important role.

Apart from the 4 Vs of big data, researchers have come up with some additional Vs. They are —

- **Variability**: This refers to the way the meaning of the data is always changing. In other words, the meaning of the data changes according to how the data is collected. One of the simplest examples of this concept would be the numerous words in the English language with multiple meanings. So, interpreting a word without considering its surrounding words will not give you its exact meaning. If you analyse the entire sentence in which the word is present, you may get the exact sense of the word.

- **Validity**: This refers to the correctness of the values present in the working dataset. The dataset is bound to have some anomalies in it, such as missing values, incorrect values, etc. It's important to get rid of these anomalies before processing this dataset. The presence of these defects will influence the output, thereby giving you erroneous results. One example of invalid data would be negative values in an age column.

- **Volatility**: It refers to determining the expiry or life expectancy of data. For better results, based on some business requirements, it's sometimes required that you archive or eliminate the old data

completely. An example of volatility would be the data corresponding to taxes such as VAT after the implementation of GST, which is archived because it will never be utilised again.

The Difference Between Veracity and Validity:

To find out how veracity is different from validity, we discussed a few examples from our day-to-day lives.

Veracity refers to the truthfulness of data. In other words, it determines whether the given data is trustworthy or not. For instance, a piece of data will suffer from veracity issues if its source is not credible. Suppose that a student is working on a case study on the 'Pollution of Rivers in India'. To create this, he/she must collect water samples from various rivers and must get them examined at a lab to determine the degree of pollution in each stream. Here, if the student has collected a single sample from every river, then you can say that the data collected suffers from veracity issues. This is because, ideally, the entire course of the river is not polluted. A river is the least polluted near its origin, and the most polluted towards the end of its path where it meets the sea. Let's assume that, in this example, the student has collected a single sample from each river at its origin. If the data is analysed, the lab readings will show that the rivers are not polluted. However, the rivers may be highly polluted after they've flown through highly populated regions. So, ideally, the data would not suffer from veracity issues if, for each river, multiple samples were collected at various points of its entire course.

Validity refers to the correctness of the data. While performing lab experiments, if properly calibrated standard instruments are not used, then the readings will be inaccurate. It can then be established that these readings are not valid.

Digitisation: One of the Major Cause for Big Data Generation

In this lecture, you learnt how digitisation has led to the exponential growth of big data. One of the first examples of digitisation was the advent of emails. With emails, message-sharing became convenient and time-saving. Hence, we still use emails for both personal and professional communications. Consequently, we've never looked back. Almost all the services around us have become digital, from booking tickets to transferring money to another bank account. Let's see how the digital data generated by a cell phone is utilised in the flow diagram below.

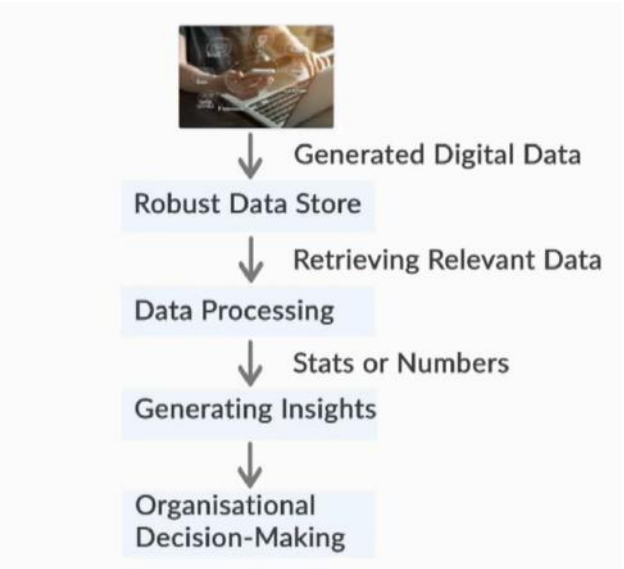


Figure 11: Storing and processing of digital data generated by cell phone

As already discussed, today, accessing the internet has become easily affordable, and internet usage has grown widespread. Each online activity, such as sending emails, booking movie tickets, uploading blogs, posting reviews on an e-commerce portal, etc. generates a massive volume of data. This growing use of the internet facilitates the generation of digital data and gives easy access to the generated data. Thus, we have entered a world that has become data-driven. Organisations and various industries leverage the power of this huge digital data reservoir to take important decisions. Additionally, the increasing usage of Internet of Things (IoT) devices has accelerated the generation of digital data.

Some of the case studies that make use of this digital data are mentioned below:

Car Digitisation:
It is assumed that all the moving parts of a car are equipped with a sensor. So, whenever the vehicle is in motion, these sensors kick in and start generating data. For example, the steering wheel, pistons in the engine, brake plates, and other sensor-fitted parts can generate data.

The data collected from these sensors gives the following information:

- The car's style of driving
- Engine temperature
- Oil temperature
- The duration the car was idle
- The car's start time

- Pedal positions

From the aforementioned information, the derived insights are —

- **Driving styles of consumers**: If the drivers are driving rash, then they can be educated about driving the right way
- Predicting the warranty costs
- Real-time health monitoring of the car

Analysing car digitisation problems in relation to the discussed Vs:

- **Volume**: Sensor data from cars across the globe
- **Velocity**: Gathering data in real time and then processing it
- **Variety**: Data is semi-structured

Sentiment Analysis Using Social Media Data:

Traditionally, after approximately six months of launching a new product, the organisation starts performing offline market surveys to collect the users' feedback.

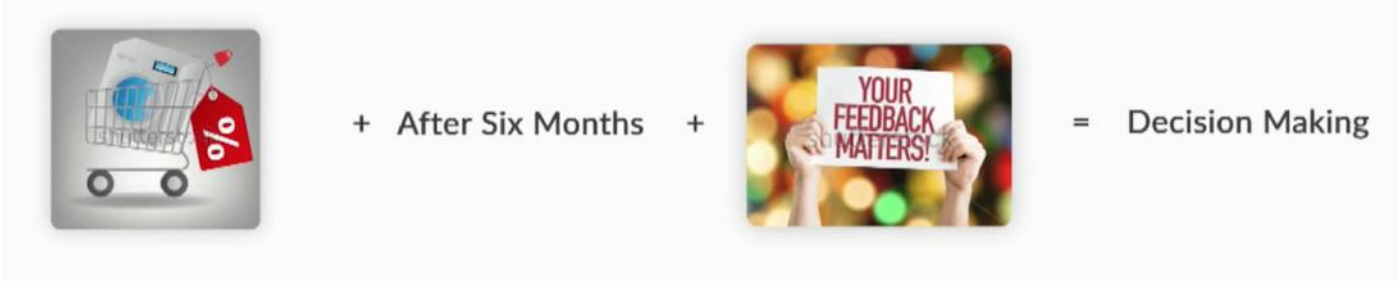


Figure 12: The older method of determining customer feedback

Today, because of the advent of social media and e-commerce websites, organisations have access to a myriad of unstructured data, which includes customer reviews, customer feedback, ratings, etc. By analysing these datasets, organisations are able to get instantaneous feedback regarding their products. Here's an image depicting how decision-making takes place in an organization:

Figure 13: A new method of determining customer feedback

Analysing the sentiment analysis problem in relation to the discussed Vs:

- **Volume**: A huge amount of social data
- **Velocity**: The data gathered is in real time and then processed
- **Variety**: The data is unstructured

Conventional Data Processing Systems and Big Data

Previously, the most commonly used data-storing and data-processing systems were RDBMSs (relational database management systems). An RDBMS uses tables to store data in a row-column format. These tables have a well-defined schema/metadata, and the data that is stored in each table must comply with the underlying schema. In most cases, operational (transactional) data is stored and analysed using RDBMSs. Storing and processing of big data is not done using traditional systems anymore.

Traditional systems fail to manage big data because of its huge size and diversity. Some big data sets are generated rapidly, because of which storing and processing such high-speed data is beyond the scope of traditional systems. Traditional systems perform well when the data is free of noise and bias. Also, sometimes the data at hand may not be credible. The lack of a mechanism to detect credibility renders traditional systems useless for the handling of big data.

Some of the typical reasons why traditional systems fail in handling big data are —

- **Types of data**: Traditional databases only support structured data. Today, big data also includes semi-structured and unstructured data, which cannot be processed using traditional systems.
- **Volume of data**: Traditional systems cannot store and process massive volumes of data efficiently. Big data processing systems store data in distributed file systems, which lead to efficient storage and processing of data.
- **Scaling**: Big data processing systems follow the scale-out architecture and distributed computing for data processing. Thus, the load of computation is shared among multiple systems. This is not possible in the cases of traditional systems because they run on single servers.
- **Data schema**: Traditional databases follow a strict schema for the data. Big data is first stored in a raw format, and then a schema is applied on it while reading it.

Lecture Notes

Introduction to Spark

In this session, you were introduced to another data processing framework named Spark. Spark is an in-memory data processing engine which processes data by holding all the data in RAM and then processing it. This session commenced by differentiating in-memory processing from disk-based processing systems like Map Reduce. As a part of this module, you were introduced to architecture and various components of Spark ecosystem.

Going forward you were also introduced to RDDs, which are the core data structures of Spark for holding data in-memory in a distributed manner. Apart from RDDs, you were introduced to Dataframes and Datasets which are used to process data using SQL like constructs.

Disk Based Processing System

In this session, you learnt that Map Reduce is a disk-based data processing engine which extensively used HDFS for processing data. In other words, all data whether it is the intermediate output or the final output are stored in the HDFS. When we say that data is stored in HDFS, it means, internally data is stored in the hard disks of the data nodes of the Hadoop cluster. So, whenever a map reduce job reads data from HDFS, it reads data present in the hard drive of the data nodes and similarly when the map reduce job produces any output the output is again written in the HDFS, i.e. hard disk of the data nodes. This means a map reduce task performs many disk I/O operations during its entire execution cycle. Writing or reading data into hard disks is more time consuming than performing read/write operations on the data which is stored in RAM.

Some of the delays encountered while accessing data from a disk are:

- **Seek time**: The time required to move the read/write head from its current position to a new position (or track) within a disk

- **Rotational delay**: The time taken by a disk to rotate under the read/write head till it points to the beginning of the data chunk to be read
- **Transfer time**: The time taken to read a complete sector or a data chunk.

The seek time contributes mostly to the latency of disk read/write operations. This means that the lesser the seeks, the better the performance. For minimising the number of seeks, Hadoop prefers reading fewer large files over many small files. Frequent movement of the disk pointer is required when there are multiple small files in the same data set. Also, Hadoop supports sequential or streaming data access instead of random access. This means Hadoop only seeks to the beginning of the file or data block and from there the data is read sequentially thereby avoiding further seeks.

Hadoop provides enormous throughput but it does not guarantee instantaneous delivery of the desired output. Throughput can be defined as the number of tasks completed in unit time or within a specific period. Some factors which aid to delayed response in Hadoop are:

- The data is always read from a disk-based file system named HDFS, hence, reading or writing data to disk is always time-consuming when compared to reading or writing data to RAM.
- Hadoop is designed to read the entire dataset even if there is a requirement to analyse only a portion of the data.
- To avoid disk seeks Hadoop encourages sequential access of data. Random access to data is not possible even if there is a need for accessing only one record present in the middle of a file. Hadoop always scans the file from the beginning.

Today due to digitisation, data volume is increasing at a very rapid rate. Also, with the increased popularity of web-based applications like search engines, social networking websites, e-commerce websites etc. the need for providing instantaneous response to user requests has become critical. The systems in the backend need to crawl through massive volumes of data and return the required information in very less time. Hadoop is a saviour when it comes to storing and managing vast volumes of data. But Hadoop is not the preferred solution when we need results instantaneously. Hadoop is more suited for batch processing, i.e. Hadoop processes historical data which is stored over a period of time. So while processing data in batch mode using hadoop it may take minutes or even hours for one job to complete. Because of these

limitations of a disk-based processing system such as Hadoop, there was a need for a different technology which can process data in real time or can return results instantaneously after crawling huge volumes of data.

In-Memory Processing and its Benefits

As Hadoop is more suited for batch processing, there was a need for different techniques that could process data in real time and generate immediate results. Hence, in-memory processing came into existence, which can load an entire dataset into memory and perform analytics on the data present in memory. In-memory processing achieved tremendous performance gain over disk-based processing by minimising the number of disks I/O operations.

When compared to disk, RAM is closer to the processor, so processes can access data randomly and faster than accessing the same data from hard drive. If the entire data set is loaded into the RAM and it stays in RAM till the processes are active and running, then instead of the disk the processes can directly read data from RAM and process it. If data is accessed from RAM all the time instead of the drive, then a significant amount of time is saved and thereby increasing productivity or response time.

Advantages of in-memory processing are:

- In-memory processing systems due to its ability to provide a quick response to a query, it is used in real time processing of data.
- In-memory processing is preferred when a query is interested in a single row out of millions of row that are available in the dataset. Required data can be directly fetched instead of scanning the entire data set.
- In-memory processing is well suited for iterative applications where the task is to process data again and again. All the intermediate output generated is also stored in memory saving us from time-consuming disk I/O operations.

UpGrad

Introduction to Spark and its Architecture

In the introduction, professor stated that Spark started as a research project in UC Berkeley in 2009. Researchers were earlier working on Hadoop MapReduce and were not satisfied with the performance of MapReduce framework for iterative and interactive computing tasks. Hence, to overcome the disadvantages of MapReduce researchers developed Apache Spark in 2014. Like MapReduce, Spark is also designed to process data in a distributed manner.

The two major reasons why Spark has gained popularity today are —

- **Speed**: Due to in-memory processing, Spark processes data with lightning speed.
- **Generality**: Spark can be used for a variety of tasks such as batch, interactive, streaming, etc. It also supports APIs in various languages such as Python, Java, Scala, etc.

Some common use cases where spark is used predominantly are:

- A spark driven application gathering streaming data in real time, filters and classifies it using some machine learning algorithms.
- Using SQL queries to join the data generated in previous use case with unstructured log files for storage purpose

Like Hadoop, Apache Spark also follows a master/slave architecture. There are daemons dedicated to the master and slave. The daemon corresponding to the master runs on the master node, and similarly, the daemon corresponding to the slave runs on the slave node. A cluster manager is an external service responsible for acquiring resources and allocating them to Spark jobs. The roles and responsibilities of the master, slave, and cluster manager are given below in detail—

Spark Master

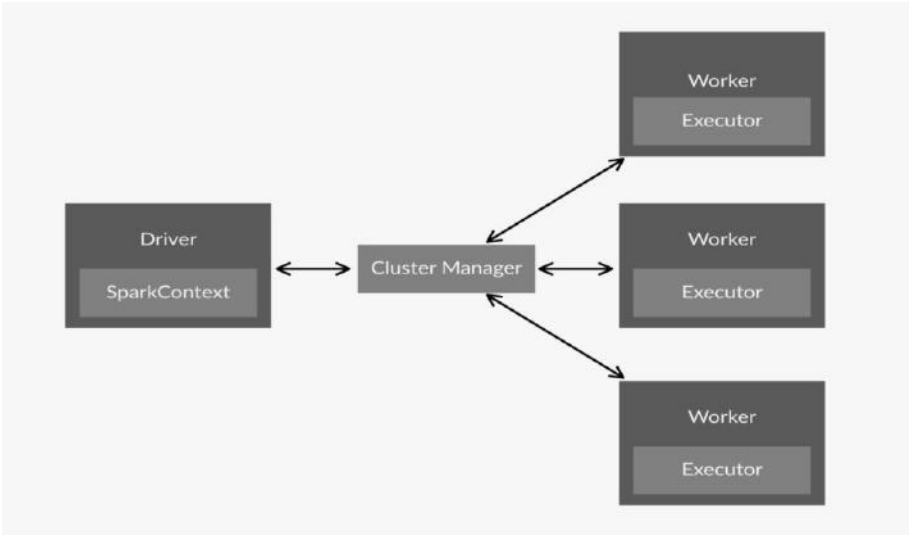
- The driver program runs on the master node.
- The driver is responsible for creating a SparkContext. However big or small the Spark job, a SparkContext is mandatory to run it.
- The SparkContext generates a directed acyclic graph of different transformations performed on the RDD.
- The driver program splits the graph into multiple stages which are then assigned to the worker nodes via the cluster manager, to process data.

Spark Slave

- Slaves are the nodes on which executors run.
- Executors are processes that run smaller tasks that are assigned by the driver.
- They store computed results either in memory, cache or on hard disks.

Cluster Manager

- The cluster manager is responsible for providing the resources required to run tasks on the cluster.
- The driver program negotiates with the cluster manager to get the required resources, such as the number of executors, the CPU, the memory, etc.



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Mapreduce vs Spark

In this session, our industry expert explained in detail the difference between Spark and Map Reduce. The most prominent advantage of using Spark over MapReduce is, Spark can do batch processing like Hadoop and real-time streaming, unlike Hadoop. Spark has the potential of increasing the processing speed to 100 times more than that of MapReduce. Spark does all its computation by storing all the data in memory, i.e. the RAM of all the nodes present in the cluster.

Another major advantage of using Spark is managing Spark is pretty easy as compared to MapReduce. For tuning and tweaking MapReduce conveniently, there is a need for various third-party tools like Embody or Ganglia whereas Spark has it own UI called SparkUI. SparkUI is used to manage the jobs. Some of the operations which can be performed using SparkUI are given below:

- Using Spark, you can find out the time of job submission, time of job completion or the time when the job failed.
- Killing the job.

Also, MapReduce is not designed to handle multiple queries at a time. Let's say there is a situation where 20 people want to query some data which is sitting on HDFS, and everyone starts putting up their jobs. A point to be noted over here is, withholding so many MapReduce jobs is difficult. Hence there are two ways of handling such situations:

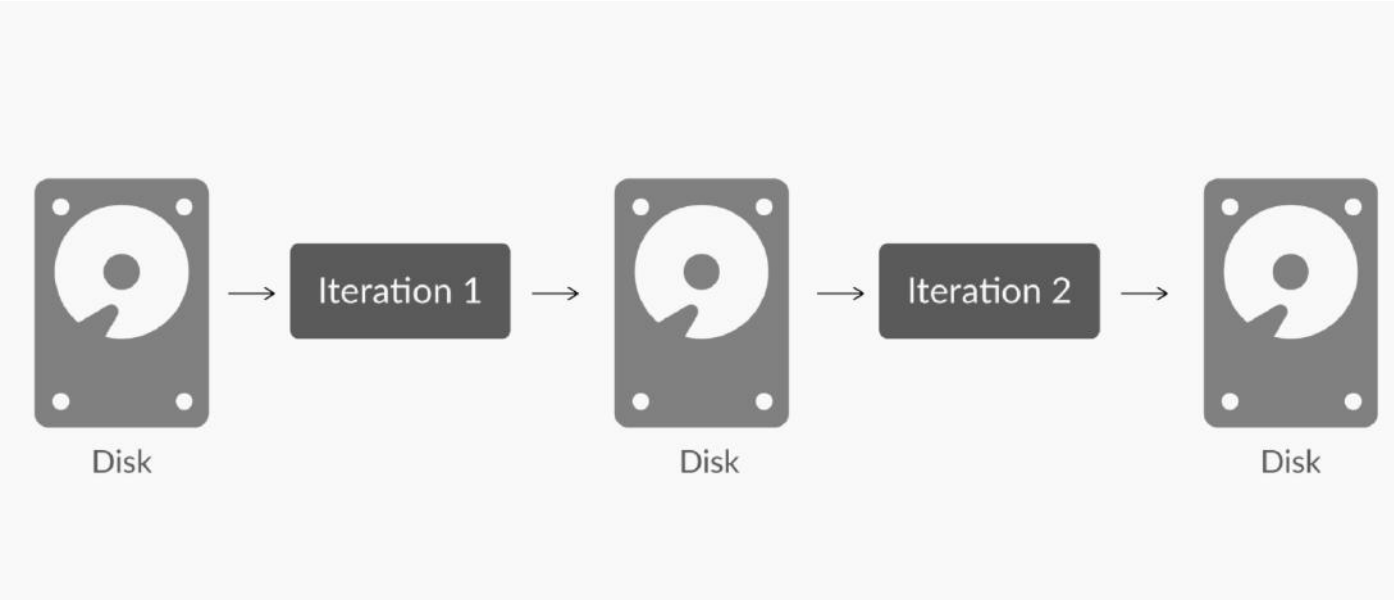
- Jobs are killed by the processing framework.
- Jobs are queued.

The first method works but is not a preferred workaround. The second solution can lead to a deadlock or a dead state because MapReduce jobs are very slow and the execution of the last job in the queue may get impossible. All these problems are solved with Spark because queuing time for a spark job is less compared to MapReduce as the job execution speed is much more than that of MapReduce jobs.

Lastly, the processing and analysis of big data help businesses in decision-making. Any decision is useful if it is taken on time. MapReduce is inherently slow because of enormous disk I/O operations. So, processing huge amounts of data using it sometimes consumes a lot more time than required. To complete data-processing tasks before their deadlines, Spark is preferred over MapReduce; because of its ability to process data in memory, it ensures low-latency data processing and retrieval. MapReduce is also not the ideal solution for processing data in an iterative manner. Let's understand this in detail:

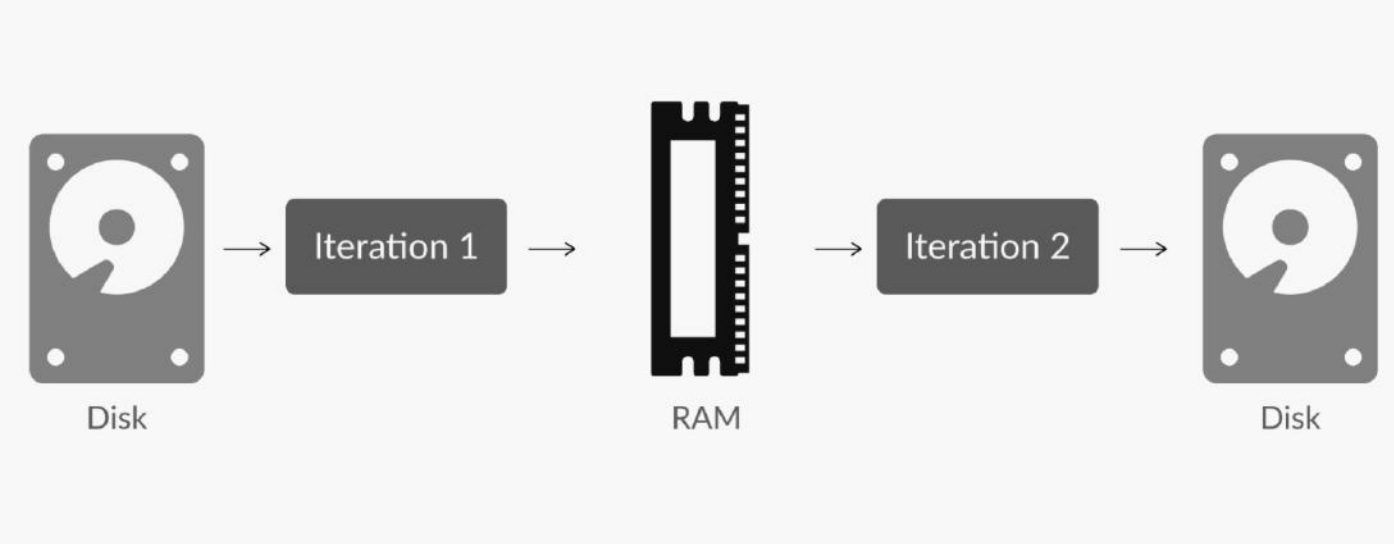
Iterative Processing Using MapReduce

- An iterative task is one that uses a chain of jobs to determine the final output. In this set-up, mostly, the child job consumes the output of the parent job.
- Now, let's assume there is a chain of two jobs, where the output of the first job is the input for the second job, and the output of the second job is the final output.
- In MapReduce, the output of the first job is stored on a disk. As this output is the input for the second job, the second job reads this data from the disk.
- For a chain of two jobs, as the intermediate data is stored on the disk, there are four disk I/O operations that generate the final output. Similarly, if there are three jobs in the chain, there will be six disk I/O operations. Always remember that disk I/O operations are costly.



Iterative Processing Using Spark

- The intermediate output is stored in the RAM instead of on the disk.
- If there are two tasks, the first one stores its output in memory, and the second reads it from memory.
- This approach reduces the total number of disk I/O operations to two.



Components of Spark Ecosystem

The Spark ecosystem comprises the following:

Spark Core(or Apache Spark)

- Spark Core is the central processing engine of the Spark application.
- It acts as an interface between Spark components and the components of a big data processing architecture such as Yarn, the HDFS, etc.
- It provides an execution platform for all Spark applications.

- Spark Core makes use of a data structure known as an RDD; this is a Resilient Distributed Dataset for performing in-memory transformations on a collection of huge data.
- Some of the other critical responsibilities of Spark Core are memory management, fault recovery, and the scheduling and distribution of jobs across worker nodes.

Spark SQL

- Spark SQL allows users to run SQL queries on top of Spark.
- With it, the processing of structured and semi-structured data is quite convenient.
- Spark SQL also introduced data frames and datasets that could impose a schema on RDDs, which are schemaless.

Spark Streaming

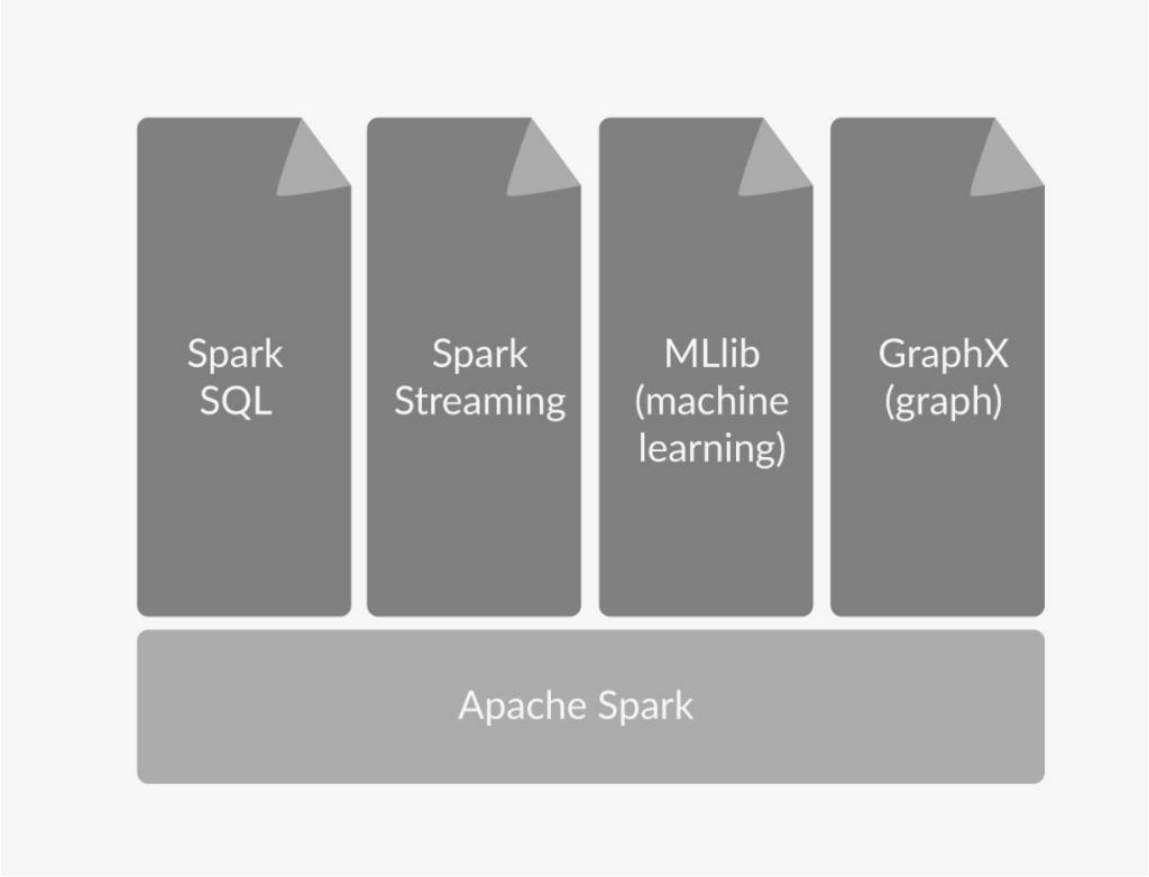
- The Spark Streaming module processes streaming data in real time. Some of the popular streaming data sources are web server logs, data from social networking websites, such as Twitter feed, etc.
- Behind the scenes, the Spark Streaming module receives input data streams and divides them into mini-batches. Each mini-batch is processed independently, and the final stream of output data is also generated in batches.
- The Spark Streaming API resembles the Spark Core API, which makes the life of a developer more comfortable as he/she works on batch processing and streaming data at the same time.

Spark MLlib

- MLlib is a machine learning library that is used to run machine learning algorithms on big data sets that are distributed across a cluster of machines.
- Due to in-memory processing in Spark, iterative algorithms such as clustering take very little time when compared to other machine learning libraries such as Mahout, which uses the Hadoop ecosystem.

Spark GraphX

- The GraphX API allows a user to view data as a graph and combine graphs with RDDs.
- It provides an API for common graph algorithms such as PageRank.



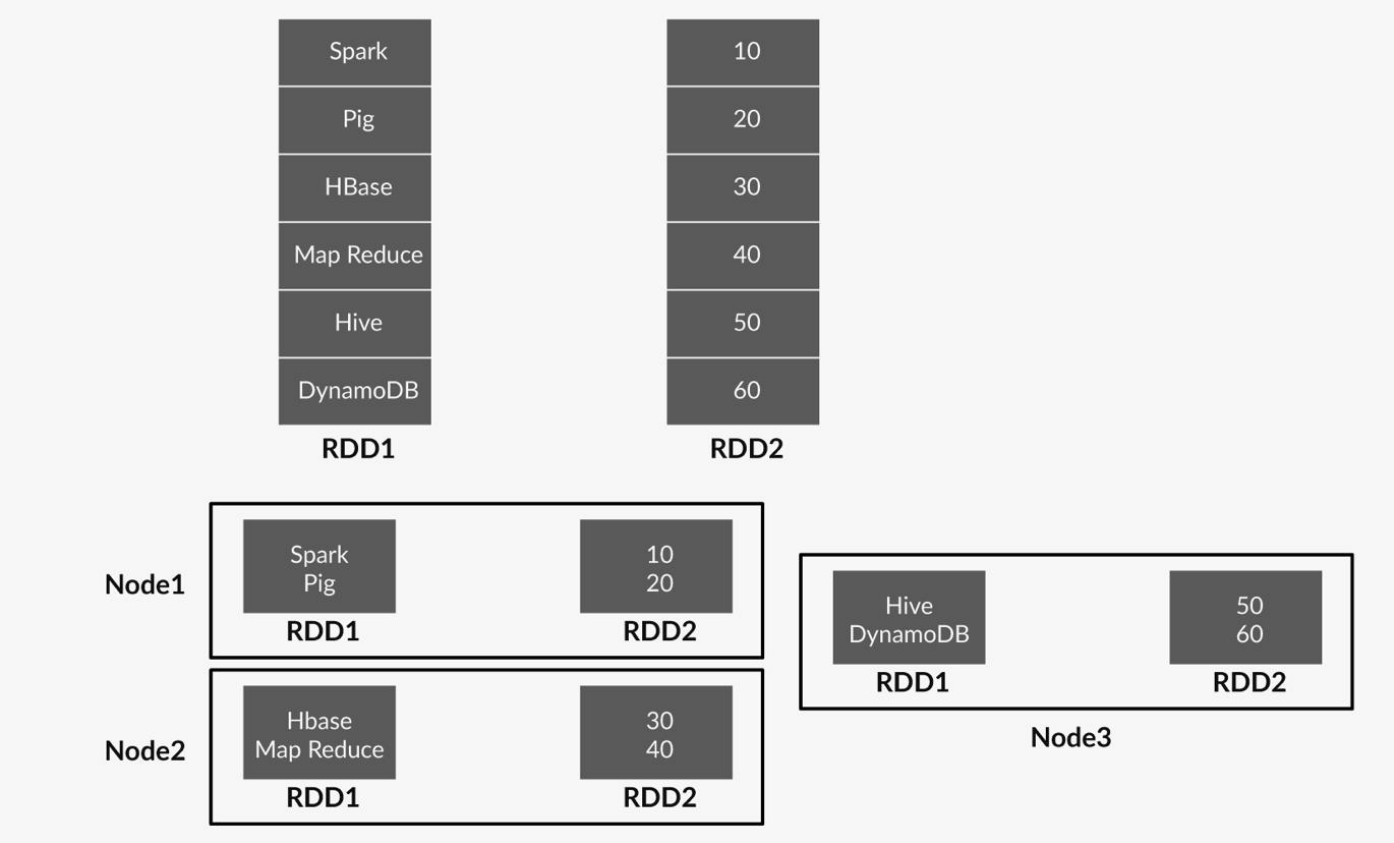
Introducing RDDs

According to the first white paper released for RDDs, an RDD is a distributed memory abstraction that lets programmers perform in-memory computations on large clusters, in a fault-tolerant manner. The acronym 'RDD' can be expanded as Resilient Distributed Dataset. An RDD is Resilient because its immutable which means the state of an RDD cannot be changed. The immutability of an RDD makes it highly fault tolerant. The data contained in an RDD is partitioned or distributed across multiple nodes.

So, some important features of an RDD are as follows:

- The term 'RDD' is expanded as Resilient Distributed Dataset, where 'Resilient' refers to fault-tolerant, 'Distributed' refers to data that resides across multiple interconnected nodes, and 'Dataset' is a collection of partitioned data.
- RDDs are Spark Core's abstraction for working with data. In other words, an RDD is a core object that is often used to deal with data in Spark.
- An RDD is a fundamental data structure of Spark that is immutable and read-only.
- RDDs can include any Python, Java, or Scala objects, or even user-defined classes.
- Each dataset is divided into logical partitions that may be computed on different nodes of a cluster. Then, under the hood, Spark distributes the data contained in the RDD internally, among the clusters, and parallelises the operation that you perform on it.

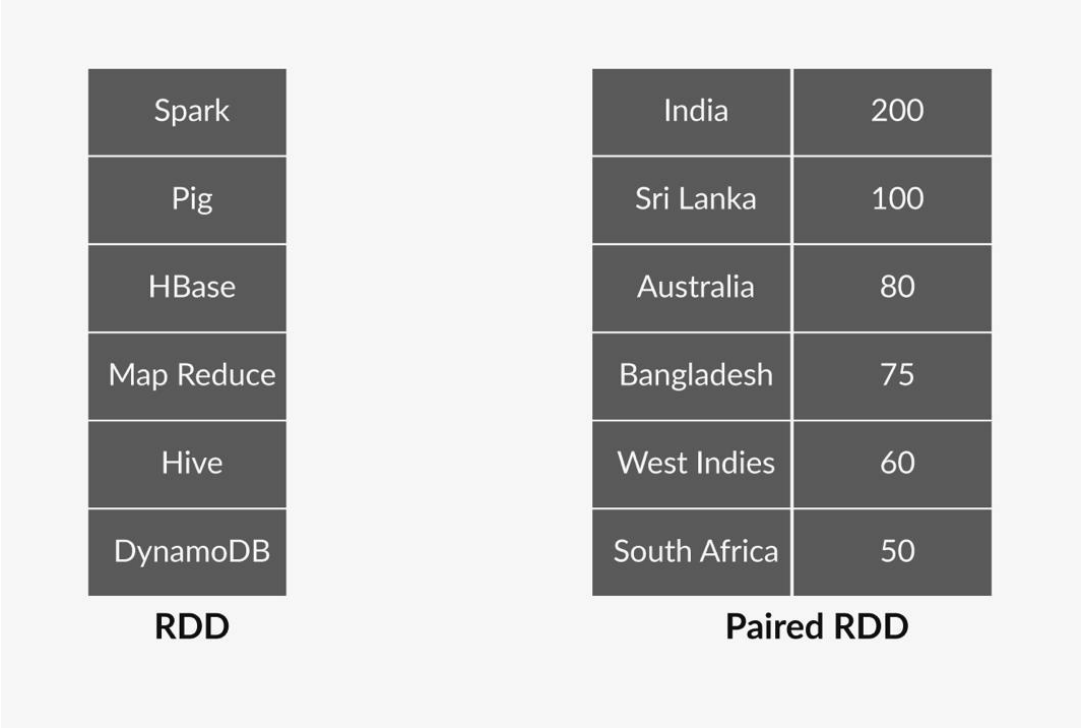
The following figure demonstrates how the data contained in an RDD is distributed across multiple clusters.



In the figure above, RDD1 contains all the string objects, and RDD2 contains all the integer objects. If the Spark cluster comprises three worker nodes, Node1, Node2, and Node3, the figure above clearly demonstrates how an RDD partitions all the data and distributes the partitions among the worker nodes.

There are two types of RDDs:

- **Basic RDDs:** These treat all the data items as a single value.
- **Paired RDDs:** These treat all the data items as key-value pairs.



In the figure above, you can see that the first RDD is a basic one, while the second RDD is a paired one in which Key is the country name of the string type and the value is of the integer type.

Operations Performed on RDDs

Operations performed on RDDs are divided into two categories:

- Transformations
- Actions

Transformations are operations or functions which operate on one RDD and generate another RDD as output. A new RDD is always created whenever a transformation is performed on an existing RDD. **Actions** on the other hand, when performed on an RDD, do not produce a new RDD as output. The output generated by the Actions can be stored in drivers or external storage systems.

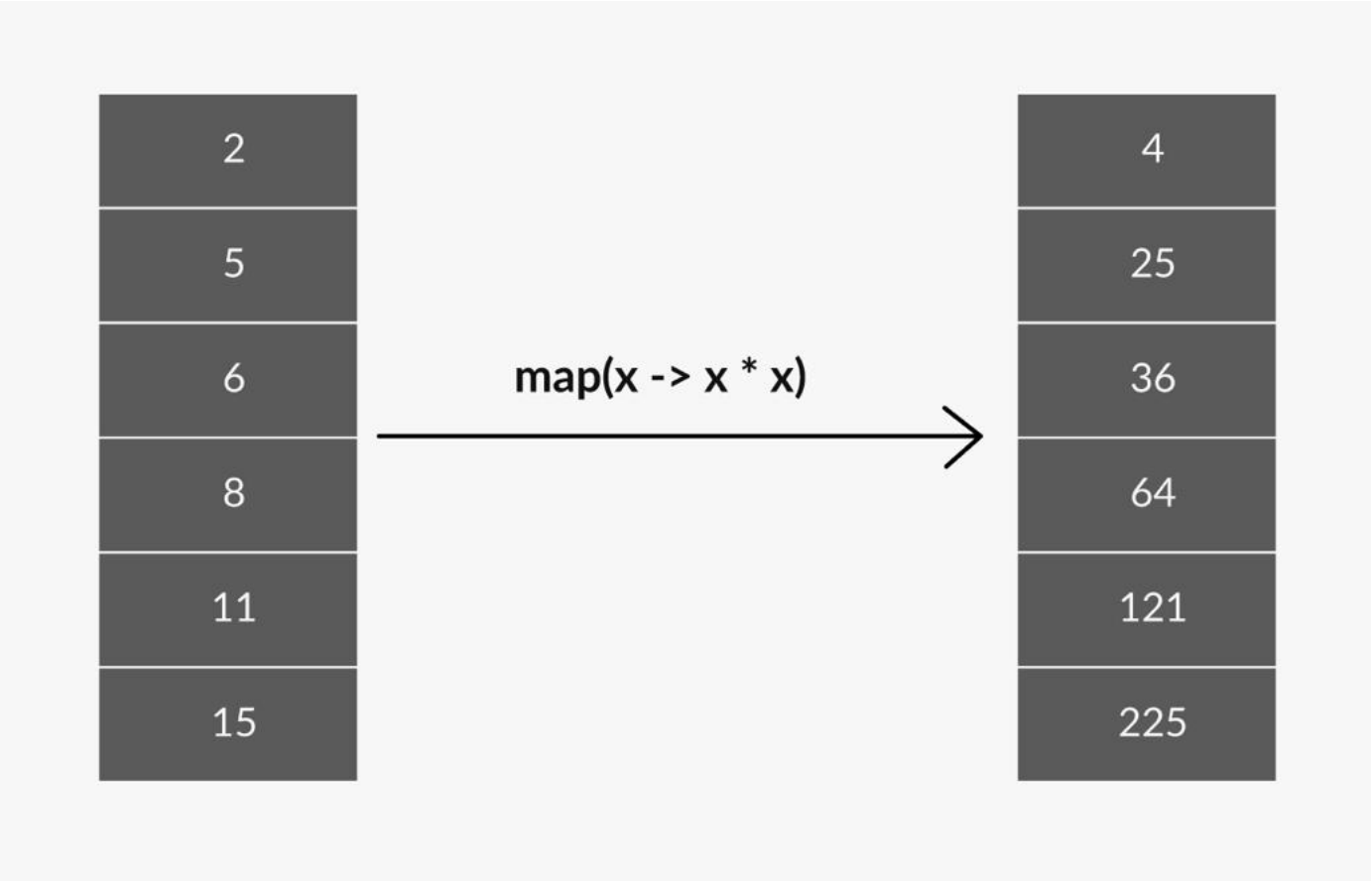
Let's say there is a sales dataset of past ten years, which is a vast data set. From this previous sales dataset, it is required to find the year in which sales are maximum in last ten years. For doing this, first read the entire dataset into an RDD and similar to SQL perform a group by operation using year, find the sum of sales and then order the output and unless an action is performed on an RDD, the results are not generated. Some of the commonly used actions are action to save the output in external storage, actions to display the content of RDD etc. A point to be noted over here is Spark is lazy in nature which means all the series of transformations performed on a parent RDD are only executed if the series of transformations ends with an action.

So, a single transformation or a series of transformations has to be followed by an action; unless an action is encountered, a transformation won't be triggered. This is known as 'lazy evaluation'.

Some common transformations that are applied to a basic RDD are —

Map

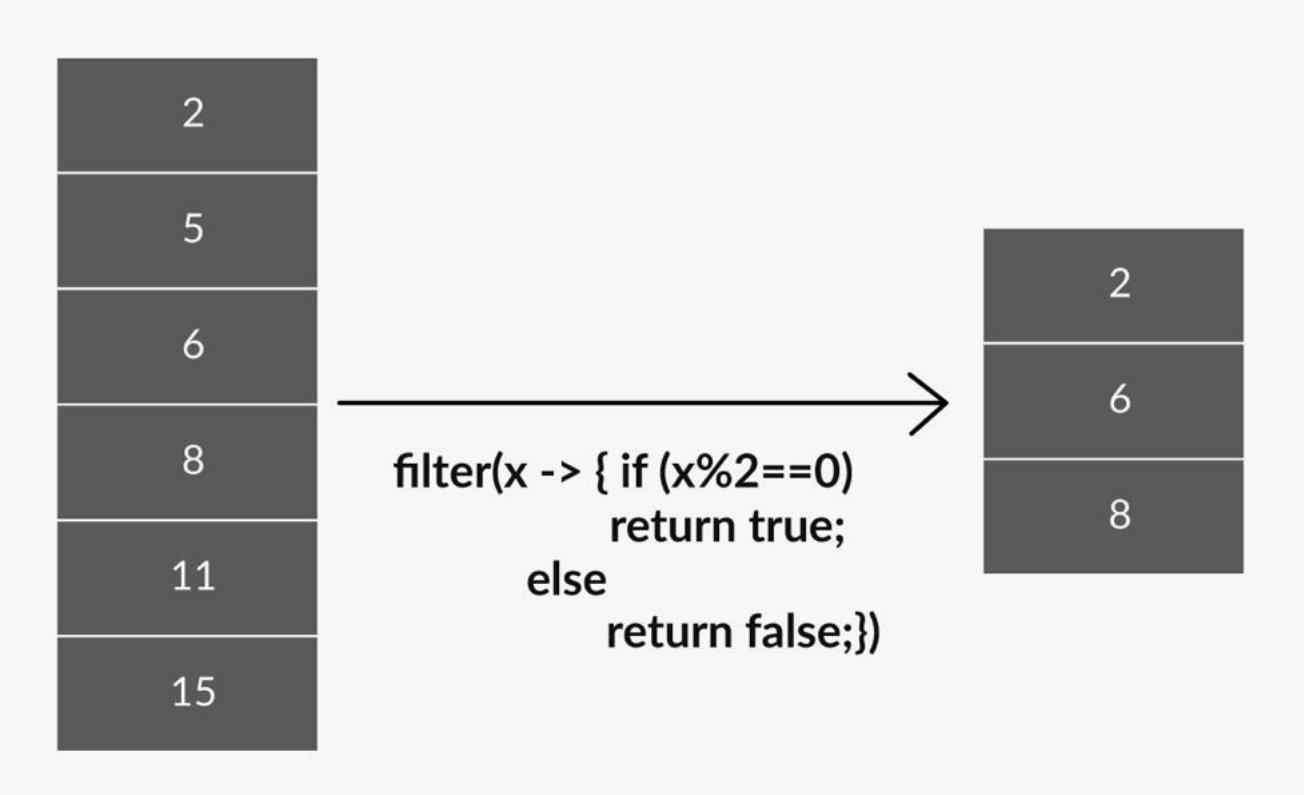
- There is a one-to-one mapping between the elements of the source RDD and those of the resultant RDD.
- A Map transformation takes a function or a lambda expression as an argument, and this function is executed on each data item of the parent RDD.
- The return value of the function when applied to each data item of the parent RDD forms the data items of the resultant RDD.



In this image, there is an RDD that has integral values in it. The Map transformation takes a lambda expression as an argument that is executed on each element of the parent RDD and returns the square of each element.

Filter

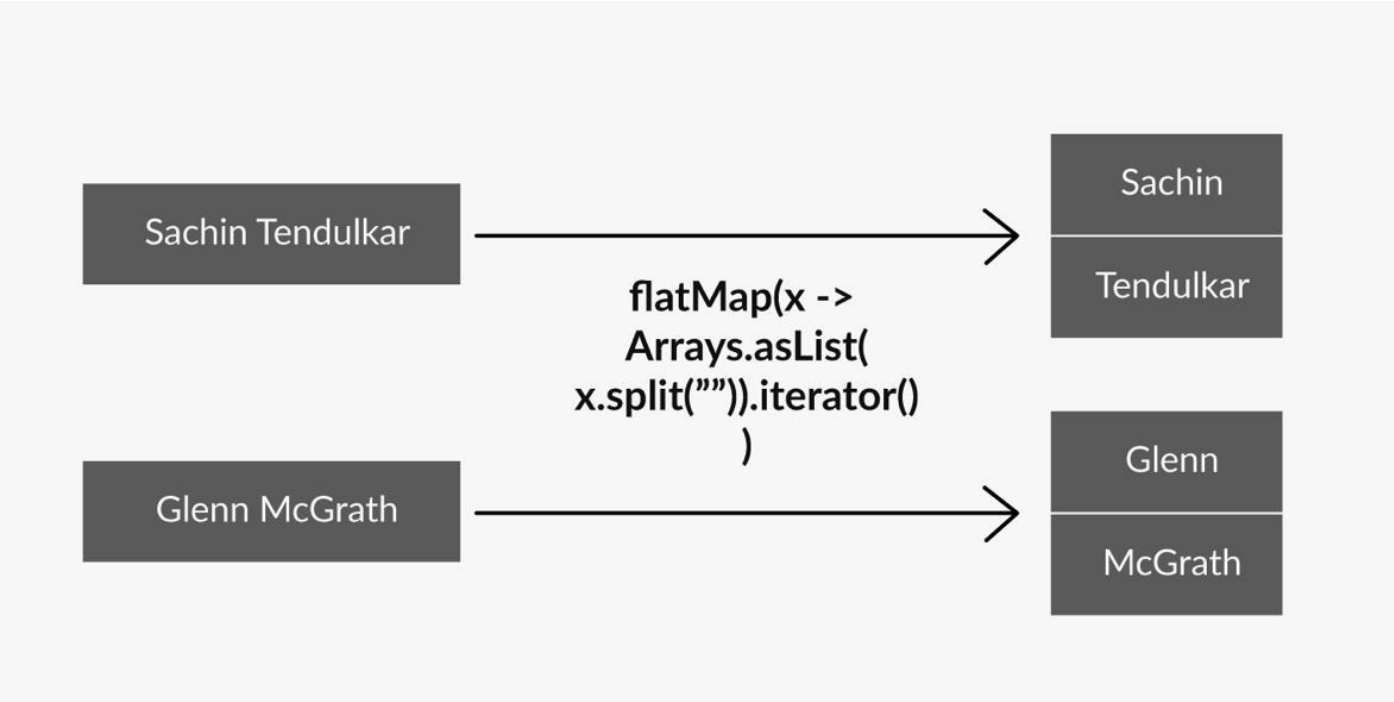
- The Filter transformation takes a function or a lambda expression that returns a boolean value as the argument.
- This boolean function is executed on each element of the source RDD.
- Only those elements that returned 'true' when the lambda expression was executed on them find a place in the resultant RDD.



In this image, there is an RDD that has integral values in it. The Filter transformation takes a lambda expression as an argument that checks if a number is even or not. If the number is even, then the lambda returns 'true'; otherwise, it returns 'false'. So, the resultant RDD will only have values divisible by 2.

flatMap

- The flatMap transformation maps an element of the input RDD to one or more elements in the target RDD.
- Unlike Map, for a single element in the input RDD, the lambda expression will return a collection of values.



In this image, the input RDD contains two string data items, i.e. Sachin Tendulkar and Glenn Mcgrath. On the application of a flatMap transformation, the final RDD will have Sachin, Tendulkar, Glenn, and McGrath as individual data elements in it.

Some common actions that are applied to a basic RDD are —

Collect

- This action collects all the data for a particular RDD distributed across partitions.
- The collected data is returned as an array to the driver program.

Count

- Count returns the total number of elements that exist in RDDs.

First

- First returns the first element of an RDD.

Lineage in Spark

Like Pig, Spark commands are also lazy. Whenever a series of transformations are performed on an RDD they are not evaluated immediately. Whenever a new RDD is created from an existing RDD, that new RDD points to its immediate parent RDD. All such dependencies between the RDDs will be logged in a graph which is known as the dependency graph or lineage graph.

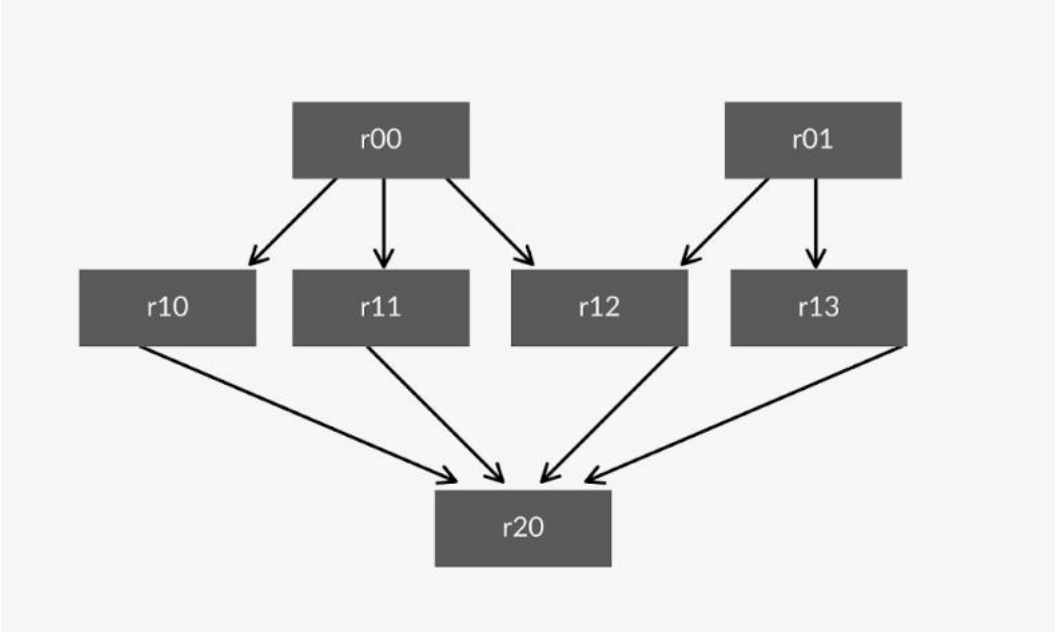
For eg: consider the below mentioned operations:

- Create a new RDD from a text file
- Apply map transformation on the first RDD to get the second RDD
- Apply flatMap transformation on the second RDD to get a third RDD
- Apply mapToPair transformation on the third RDD to generated a fourth RDD

Lineage graph of all these operations looks like:
First RDD ---> Second RDD (applying map) ---> Third RDD (applying flatMap) ---> Fourth RDD (applying mapToPair)

This lineage graph is useful if any of the partitions for an RDD is lost, the partitions are recomputed by referring to the lineage graph.

A lineage graph of all the RDDs created during a job execution is mentioned below:



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Lecture Notes

Lambda Expressions

Till version 7, Java was purely imperative. In version 8, Java included functional programming features through lambda expressions. It allowed java programmers to use functional programming with less learning effort. It saved java programmers from learning an entirely different language with altogether different language construct, grammar and syntax

This session introduced lambda expression because lambda expressions will be used extensively for developing spark programs.

Why Lambda Expressions

Let's consider a class ComputeInterest. The entire structure of the class ComputeInterest is mentioned below:

```
class ComputeInterest{
	private double principal;
	private double rate;
	private double time;

	public ComputeInterest(double principal, double rate, double time){

		this.principal = principal;
		this.rate = rate;
		this.time = time;
	}

	double computeSimpleInterest(){
		return (principal*rate*time)/100;
	}
}
```

This class has the principal, the rate of interest and time as member variables and has a method computeSimpleInterest for determining the simple interest. Let's say because of change in requirement instead of simple interest you will have to compute the compound Interest now. The only way of solving this problem is by adding an additional method named computeCompoundInterest which will have the logic for computing the compound interest. In real life scenario, repeatedly making changes in the class files, testing and deploying the same is very tedious and consumes a lot of time. One of the ways by which this problem can be solved is if we can pass a computation as a parameter to the method and executing this computation within the method. A pseudocode for the same is given below for your reference:

```
operation = square(x) // defined the operation
performOperation(operation) // passed the operation as parameter void
performOperation(Operation operation) {
     operation.perform() // executed operation
}
```

For computing cube, the only modification which has to be done in the above code is :

operation = cube(x)

Rest of the code remains same as before. This feature of passing an operation as a parameter is introduced in Java 1.8 through Lambda Expressions.

Understanding Lambda Expressions

In this session, you were introduced to the structural or syntactical aspects of a lambda expression. A Java Lambda Expression is an anonymous function which does not belong to any specific class. Lambda expressions do not require a name, return type, and type for the input parameters. Lambda expressions provide a way to pass behaviours as arguments to other functions.

The syntax of a lambda expression is:

(Parameters) -> {Body}

Let's discuss each component in detail:

- **Parameters**: Define the list of parameters passed to Lambda Expression. A lambda expression can accept any number of parameters including zero.
- **->**: Arrow-token is used to link parameters and body together.
- **Body**: Contains zero or more statements which define what a Lambda Expression will do.
- **Type**: In Java, the lambda expressions are represented as objects, and are bound to a particular object type known as a functional interface. Functional interfaces are discussed in a separate segment.

Some points to remember while writing lambda expressions:

- If the body contains only a single statement then the enclosing curly braces are not required.

  (x) -> System.out.println(x)

- It is not mandatory to declare the types of the parameters in the lambda expressions. The types are dynamically inferred during compile time.

  (x, y) -> System.out.println("Type inference in action" + (x+y))

- If a single parameter is used in lambda expression then the enclosing parentheses can be ignored.

  x -> System.out.println(x)

- If the lambda expression consists of only one statement then the return statement and curly braces can be omitted.

  x -> x * x

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Converting a Java Method to Lambda Expression

In this session, you learnt how to convert a java method to its corresponding lambda expression. Let's assume the method to be converted is:

```
public int add(int x, int y)
{
      return (x + y);
}
```

The stepwise conversion of the above method to its corresponding lambda is as follows:

- Remove the first three words i.e. public, int and add from the lambda expression. In a lambda expression the access specifier, return type and function name are not required.

```
(int x, int y)
{
      return (x + y);
}
```

- As we learnt in the previous section the parameter list and the lambda body is separated by - >. Let's add this operator.

```
(int x, int y)->
{
 return (x + y);
}
```

- Remove the data types from the parameter list as they are dynamically inferred.

```
(x, y)->
{
  return (x + y);
}
```

- As the body of the lambda expression consists of one line remove the curly braces and the return keyword.

```
(x, y)-> x + y
```

So the final lambda expression is : **`(x, y)-> x + y`**

Data Type of Lambda Expression

Lambda is an operation or a block of code which can be passed to other functions as parameters. In Java, every parameter has a type. As lambda can be passed as a parameter, it also has a type. The type of the lambda expression is defined using Interfaces. This means a lambda can be directly assigned to a reference variable of type interface.

The following code demonstrates how a reference variable of type interface holds a lambda expression without any issues.

```
interface MyLambda {
     public int add(int x, int y);
}

//The lambda can be written as follows:

public class AdditionUsingLambda {
     public static void main(String args[]) {
          MyLambda lambdaExpression = (x, y) -> x + y;

          /* The type of lambda is an interface MyLambda as defined
          above */

          System.out.println(lambdaExpression.add(5, 4));
     }
}
```

In this example, the lambda (x,y) -> x + y is assigned to a variable lambdaExpression which is of type MyLambda. MyLambda is the interface which has an abstract method “add” defined in it. The lambda primarily defines the functionality of the abstract method add. The lambda is made to run by calling the add method using the variable lambdaExpression. The types of x,y and the return type of the lambda expression is dynamically inferred from the abstract method declaration add in the interface MyLambda.

The interest problem discussed before is implemented below using lambda expressions:

```
public class UnderstandLambda {

    public void print(String s) {
        System.out.println(s);
    }

    interface Interest {

        public double calculateInterest(double p, double r, int t);
    }

    public static void main(String args[]) {
        UnderstandLambda ob = new UnderstandLambda();

        ob.print("Hello World!");
        ob.print("Welcome to the Reserve Bank of India!");

        Interest lambdaExpressionSimpleInterest = (p, r, t) -> (p * r * t) / 100;

        System.out.println("Simple Interest is " + lambdaExpressionSimpleInterest.calculateInterest(1000f, 5f, 2));

        Interest lambdaExpressionCompoundInterest = (p, r, t) -> p * Math.pow(1 + (r / 100), t) - p; // Compounded yearly

        System.out.println("Compound Interest is " + lambdaExpressionCompoundInterest.calculateInterest(1000f, 5f, 2));
    }
}
```

```
OUTPUT: Hello World!
        Welcome to the Reserve Bank of India!
        Simple Interest is 100.0
        Compound Interest is 102.5
```

But one important question which is still left unanswered is, what will happen if the interface has multiple abstract methods. If the interface has multiple abstract methods, then which method will be used by the lambda expression to infer the return type and data type of its parameter list. This ambiguity was clarified using the below mentioned example.

```
interface MathOperations {
    public int add(int x, int y);

    public int subtract(int x, int y);
}

public class InterfaceImplementaion implements MathOperations {

    @Override
    public int add(int x, int y) {
        return (x + y);
    }

    @Override
    public int subtract(int x, int y) {
        return (x - y);
    }

    public static void main(String args[]) {
        InterfaceImplementaion ob = new InterfaceImplementaion();

        System.out.println(ob.add(5, 4));
        System.out.println(ob.subtract(5, 4));
    }
}
```

The above program works fine without any exceptions and displays output as 9 and 1 in the console. The same program using lambda is mentioned below:

```
interface MathOperations {

     public int add(int x, int y);
     public int subtract(int x, int y);
}

public class MulUsingLambda  {

     public static void main(String args[]) {

          MathOperations lambda1 = (x,y) -> x+y;
          MathOperations lambda2 = (x,y) -> x-y;
          System.out.println(lambda1.add(5,4));        S
          System.out.println(lambda2.subtract(5, 4));
          }
```

```
}
```

The above program will flag a compilation error i.e. “The target type of this expression must be a functional interface”. This happened because the compiler got confused as it was unable to link the lambda expressions with the functions defined in the interface. Both the lambda expressions are in line with the function signatures of add and subtract functions defined in the MathOperations Interface. Hence, the compiler fails to distinguish between which Lambda Expression should be linked with which function signature.

Hence to avoid such ambiguities, it is mandatory to use interfaces having just one abstract method for defining the type of Lambda Expressions. Such interfaces are known as **functional interfaces**.

The characteristics of a functional interface, these are:

- The functional interface is a normal interface which is used in Java.
- The functional interface will have one abstract method.
- The annotation @FunctionalInterface is used to denote that the interface following this annotation is a functional interface.

The corrected code using a Functional interface MathOperations is given below for your reference:

```
@FunctionalInterface
interface MathOperations {
    public int operation(int x, int y);
}

public class FunctionalInterfaceImplementation {

    public static void main(String args[]) {
        MathOperations lambdaExpression1 = (x, y) -> x + y;
        MathOperations lambdaExpression2 = (x, y) -> x - y;

        System.out.println(lambdaExpression1.operation(5, 4));
        System.out.println(lambdaExpression2.operation(5, 4));
    }
}
```

```
OUTPUT: 9
        1
```

Advantages of Lambda Expressions

Some of the key advantages of lambda expressions are:

- **Reduces Boilerplate Code:** Lambda Expressions are much cleaner than the actual methods in java. With Lambda, things like declaring the data type of variables and usage of return keyword can be avoided, thus making the code compact. For example, to multiply two numbers and return the result, the lambda expression would be :

  `lambdaExpressionMultiply = (x, y) -> x * y;`
- **Enables Functional Programming:** Lambda expressions allow the behaviour itself to be passed on just like the function parameters thus enabling Functional Programming.
- **Easier to use APIs and Libraries:** With Lambda Expressions in hand, one can make better use of the APIs and Libraries of Java to iterate, filter and extract Collections Data more effectively and efficiently. You will learn this in detail when Spark transformations will be introduced.
- **Enables support for Parallel Processing:** As learnt before lambda expressions are pure and deterministic in nature. Hence, they can be easily used for parallel processing.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

UpGrad

Lecture Notes

Non-Linear Data Structures

In this module, you were introduced to tree data structures — conventional non-linear data structures that are used to deal with the queries in specific non-linear data ranges in an organised way. The first session explained to you binary search trees, which are used to store non-linear data and provide logarithmic running times for the fundamental operations even in the worst case. In the second session, you learnt about KD trees, another tree data structure that helps in the efficient storing and processing of multidimensional data.

Tree Data Structure

The non-linearity of data led to the evolution of non-linear data structures such as trees. The tree data structure connects every data point with several other data points in such a way that the tree represents specific relations between the connected points, as shown in the image given below:

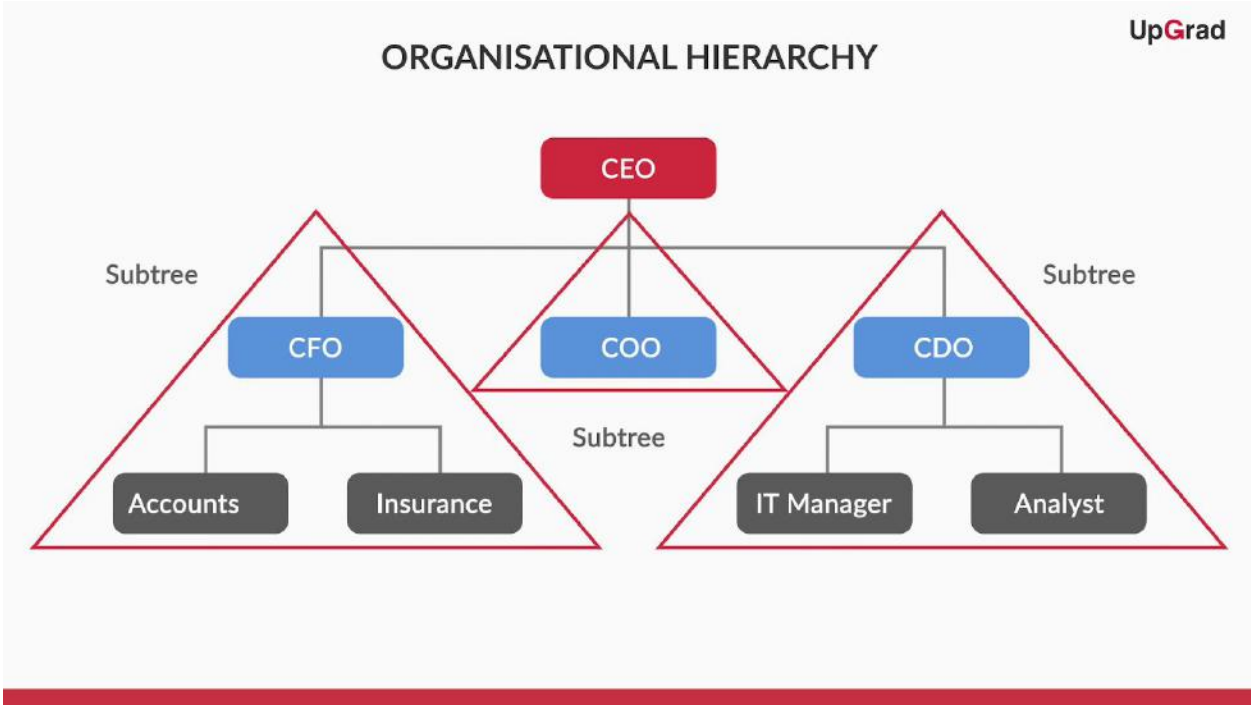


Figure 1: Organisational hierarchy

The hierarchical relationships between the points are expressed using parent and child nodes. A node that is a predecessor of any other node is known as a parent node, and a node that is descendant of any other node is called a child node. A subtree of a particular tree consists of a parent node and all the descendants of that node in the tree.

The different nodes in a tree data structure are —

- **Root node:** The node at the topmost position of a tree is called the root node
- **Leaf node:** A node without any child node is called a leaf node
- **Internal node:** The nodes that are not leaf nodes but have children are called internal nodes

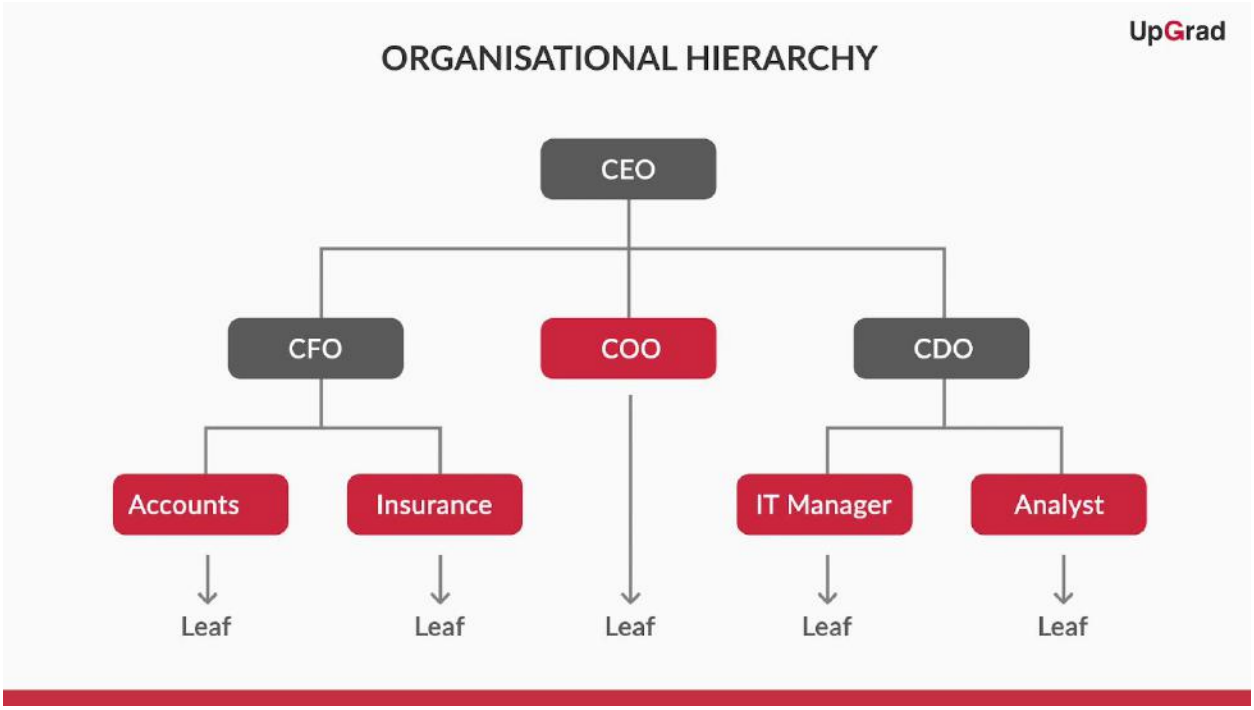


Figure 2: Organisational hierarchy

Note: Every node in a tree can have only one parent node, and the root node has no parent node.

Binary Search Trees

There are some standard tree data structures that are extensively used for handling hierarchical non-linear data.

- **Binary tree:** In this tree data structure, every node can have two children at the most. The node to the left is called the left child, and the node to the right is called the right child.

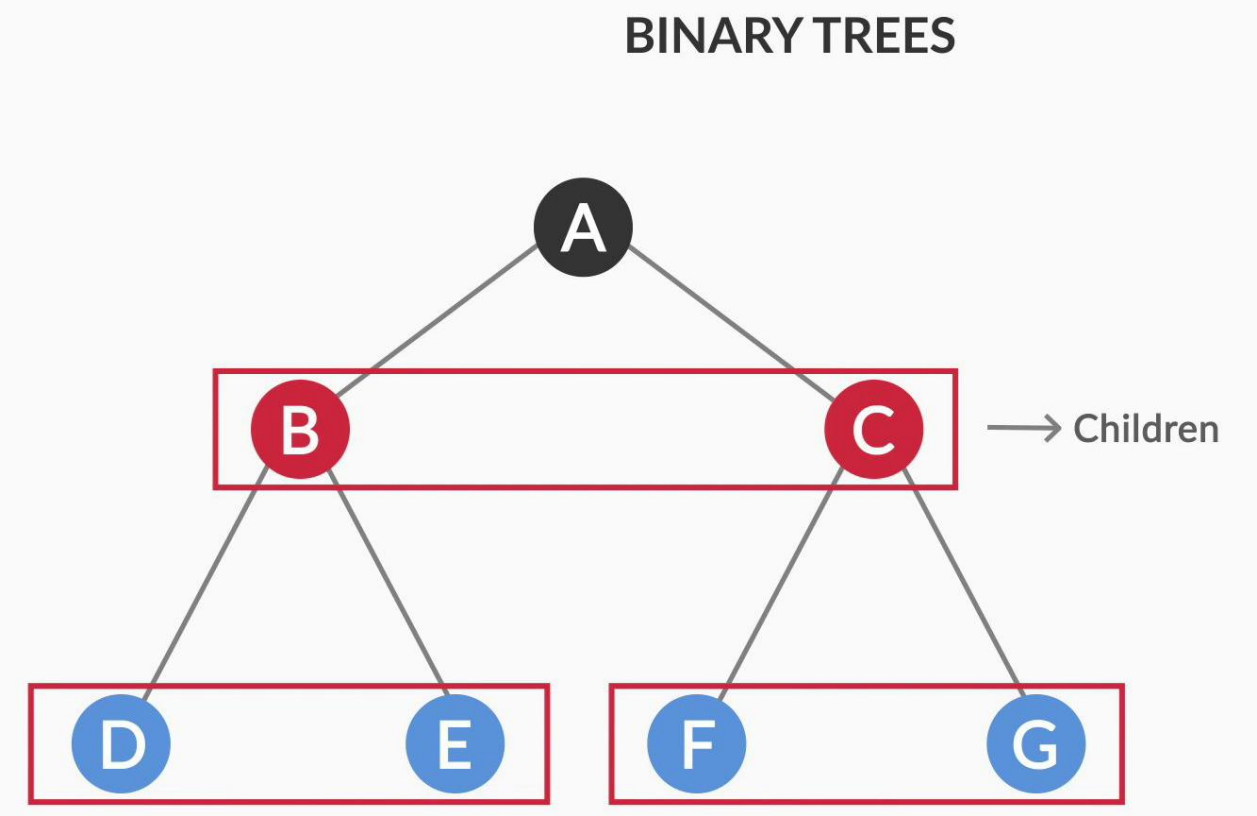


Figure 3: Binary tree

- **Binary search tree:** This is an ordered binary tree with the following specific properties:
    a. All the nodes in the left subtree are less in value than the root node
    b. All the nodes in the right subtree are greater in value than the root node
    c. Again, each subtree behaves like a binary search tree

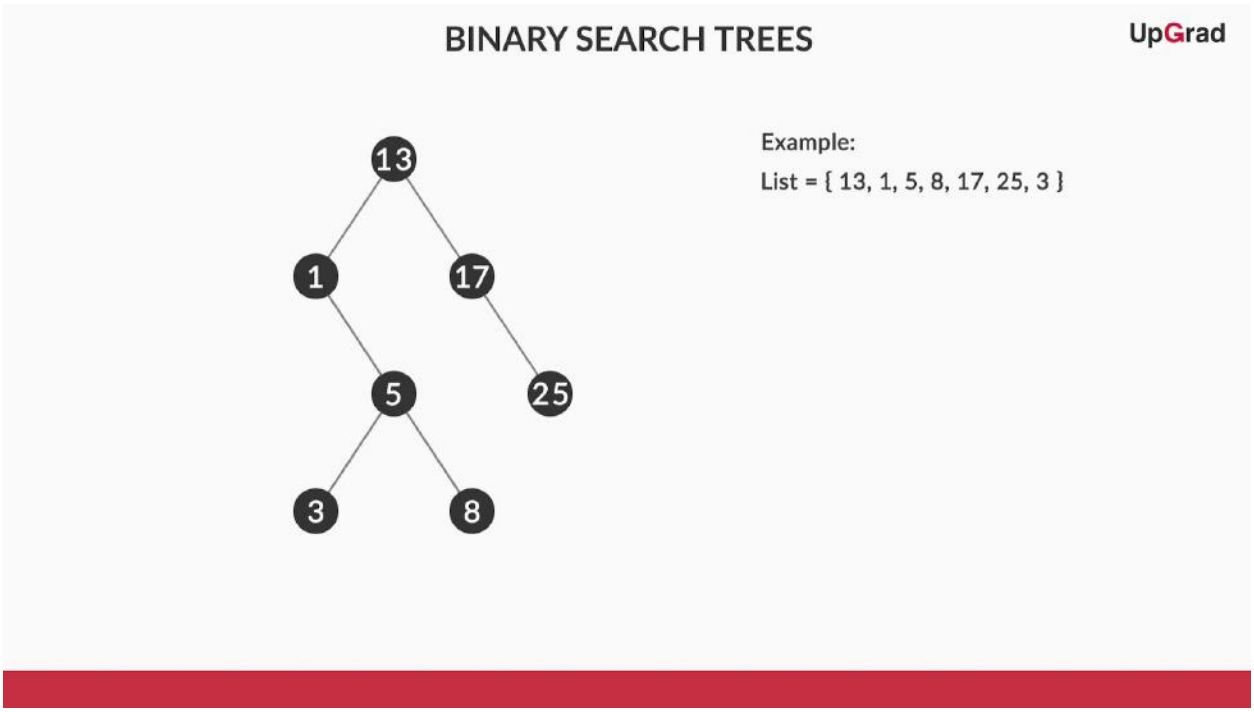


Figure 4: Binary search tree

In a search operation on a BST, the root serves as the median. A search operation is performed as follows:

1. The search key is compared with the root key, and if it doesn't match then the following steps are carried out:
    a. If the search key is greater than the median key, it traverses to the right subtree and continues the search process
    b. Or, if it is less than the median key, it traverses to left subtree and continues the search process.

This process is repeated until the search key is found or the leaf node is reached.

Note: The search process adopted in binary search trees is identical to the binary search process. Because of this similarity, it is called a binary search tree.

Optimising Search Efficiency

Apart from storage, binary search trees need to process data efficiently. Pre-processing of data before insertion can improve the search efficiency of binary search trees.

Before storing, data is sorted by choosing the median of all the elements as the root of the tree and iterating the same for all the subtrees. Storing data in this way will always produce balanced

binary search trees and, in turn, will enhance the search process efficiency as it reduces the number of comparisons in the trees.

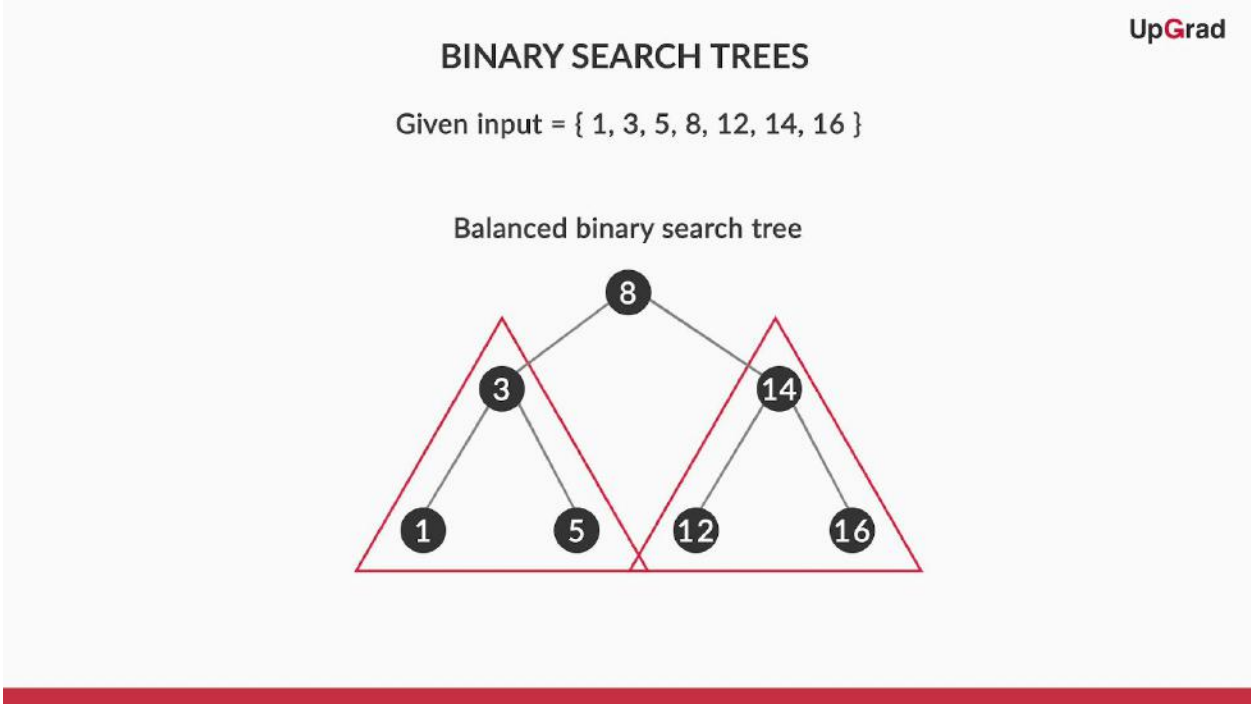


Figure 5: Balanced binary search tree

The pre-processing cost of data involves both the sorting and the inserting costs and is proportional to O(n log n). Preprocessing any given data will make the search queries cost O(log n) even in the worst case.

Tree Traversals

Contrary to linear data structures, which can be traversed only in linear order, trees can be traversed in different ways. Some of the standard ways of traversing trees are —

Inorder traversal: It is one of the typical methods used for traversing trees. An inorder traversal always produces the data in the binary search tree in ascending order.
The algorithm for inorder traversal is as follows:

1. Check if the node is empty/null
2. Visit the left subtree
3. Visit the root
4. Visit the right subtree

Preorder traversal: In a preorder traversal, the root node is visited before the left and right subtrees. The algorithm for preorder traversal is as follows:

1. Check if the node is empty/null
2. Visit the root
3. Visit the left subtree
4. Visit the right subtree

Postorder traversal: In a postorder traversal, the root node is visited after the left and right subtrees. The algorithm for postorder traversal is as follows:

1. Check if the node is empty/null
2. Visit the left subtree
3. Visit the right subtree
4. Visit the root

Note: Every subtree is considered a tree in this traversal.

Rank Queries

Binary search trees have nodes that contain comparable key values. 'Ranking' defines the position of a key in relation to other keys. The two most common rank queries performed using binary search trees are —

1. **Minimum element:** The node in the left subtree of the root that doesn't have any left child is the minimum element of the tree.
2. **Maximum element:** The node in the right subtree of the root that doesn't have any right child is the maximum element of the tree.

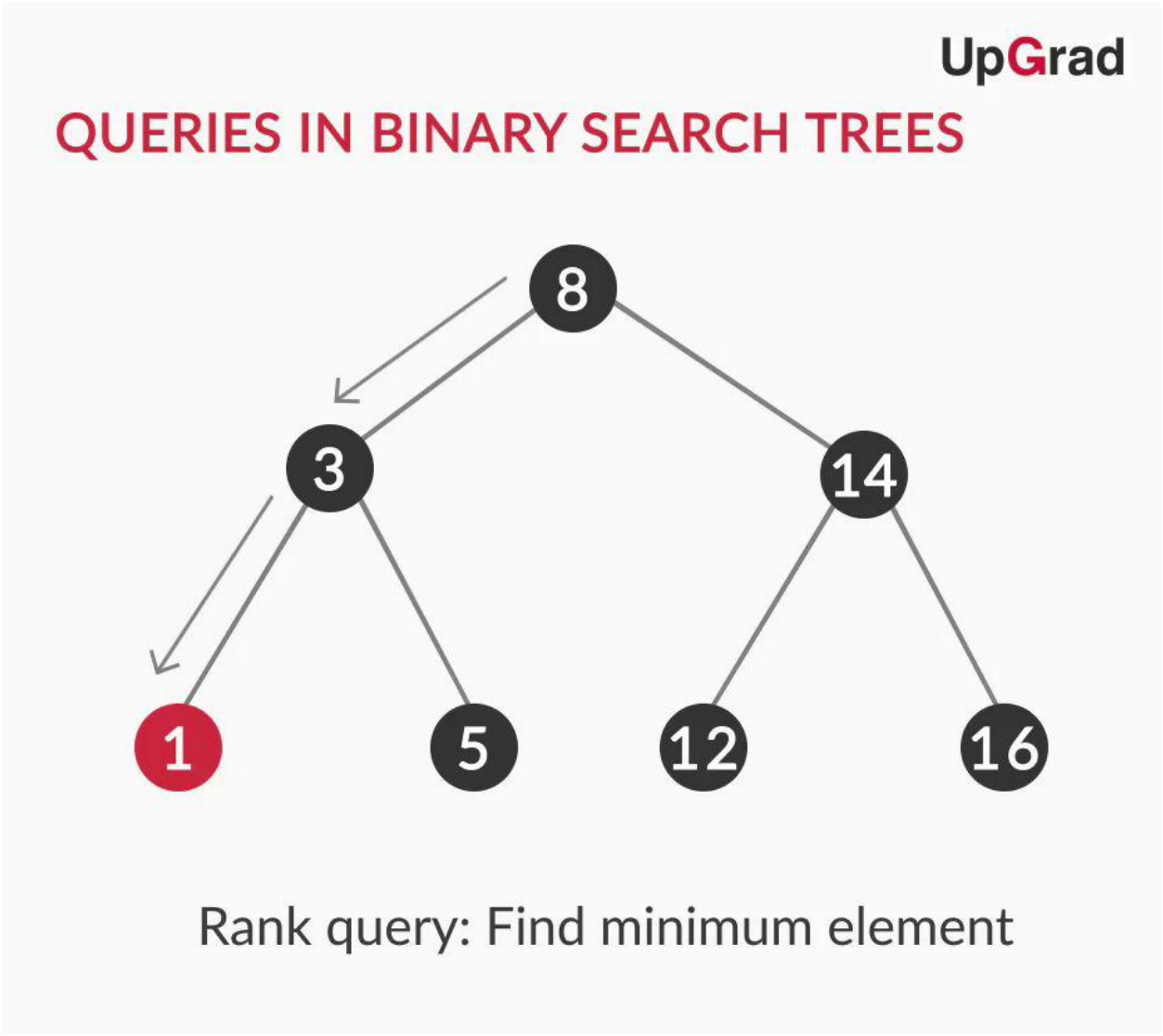


Figure 6: Rank queries

Rank queries in binary search trees are utilised in building different data structures called double-ended priority queues, which have additional applications such as external sorting. If the elements in the data, which need to be sorted, require more space than the available memory, then an external sort is implemented using a double-ended priority queue. The steps are as follows:

1) Take the elements that can fit into the memory and build a double-ended priority queue or binary search tree from them
2) After that, take the remaining elements from the data and carry out the following:
    a. If the element is less than or equal to the minimum element of the binary search tree, place it in the left set of the data.

b. If the element is greater than or equal to the maximum element of the binary search tree, place it in the right set of the data.
c. Otherwise, remove either the maximum or minimum element from the binary search tree and insert the element in its place in the tree (the deletion of a node in a binary search tree is explained in the final segment). If the maximum element is removed, then it is added to the right set of the data; and if the minimum element is removed, it is added to the left set of the data.

3) Finally, output the elements in the binary search tree in a sorted order as the middle set of the data.
4) Recursively, sort the left and right sets of the data.

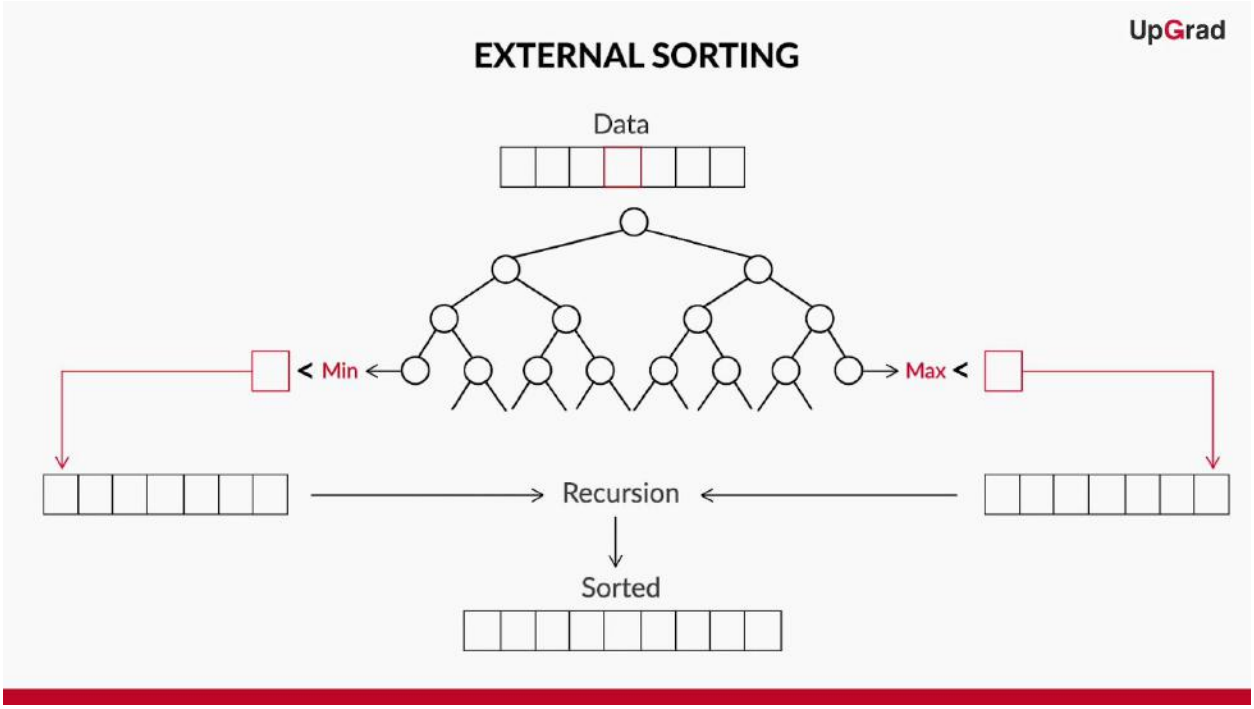


Figure 7: External sorting

Range Queries

Using range queries, you can retrieve all the keys from the data that fall within a specific range. Range queries in binary search trees are performed as follows:

1. If the node is smaller than the lower bound of the range, then it traverses its right subtree
2. If the node is greater than the upper bound of the range, then it traverses its left subtree
3. If the node is within the range, then the range is selected, and both its left and right subtrees are traversed

RANGE QUERIES IN BINARY SEARCH TREE

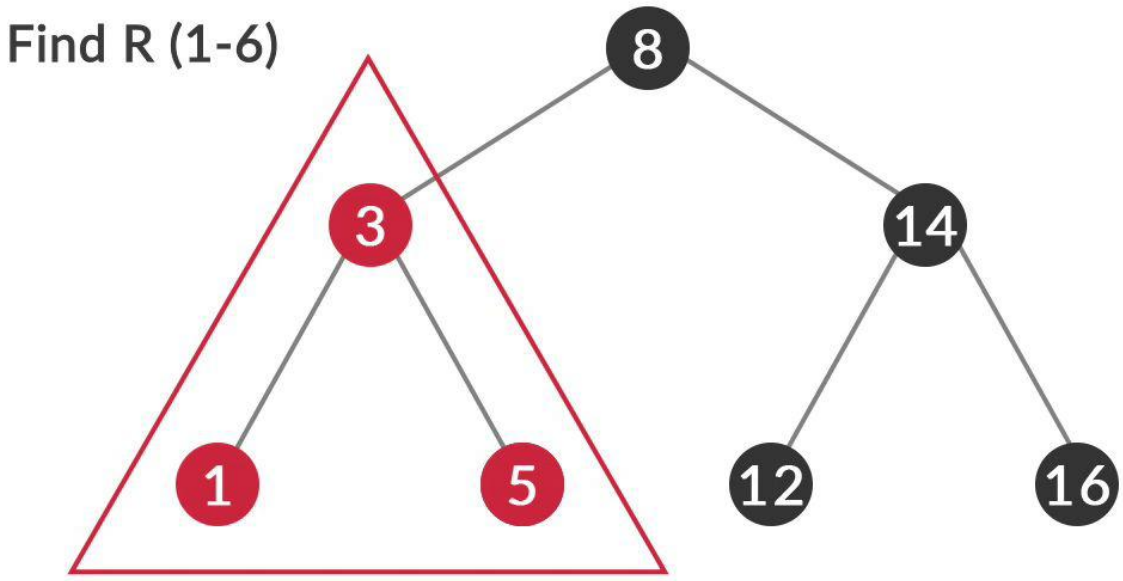


Figure 8: Range queries in binary search tree

The costs of range queries in binary search trees are as follows:

1. The cost to find the left endpoint = Log (n)
2. The cost to find the right endpoint = Log (n)
3. The cost to cover the 'K' elements in the range = K

The total cost to perform range queries in binary search trees = log (n) + log (n) + K.

Dynamic Binary Search Trees

Data, in general, is not static and is dynamic. To handle frequent modifications, both insertion and deletion operations frequently occur in binary search trees.

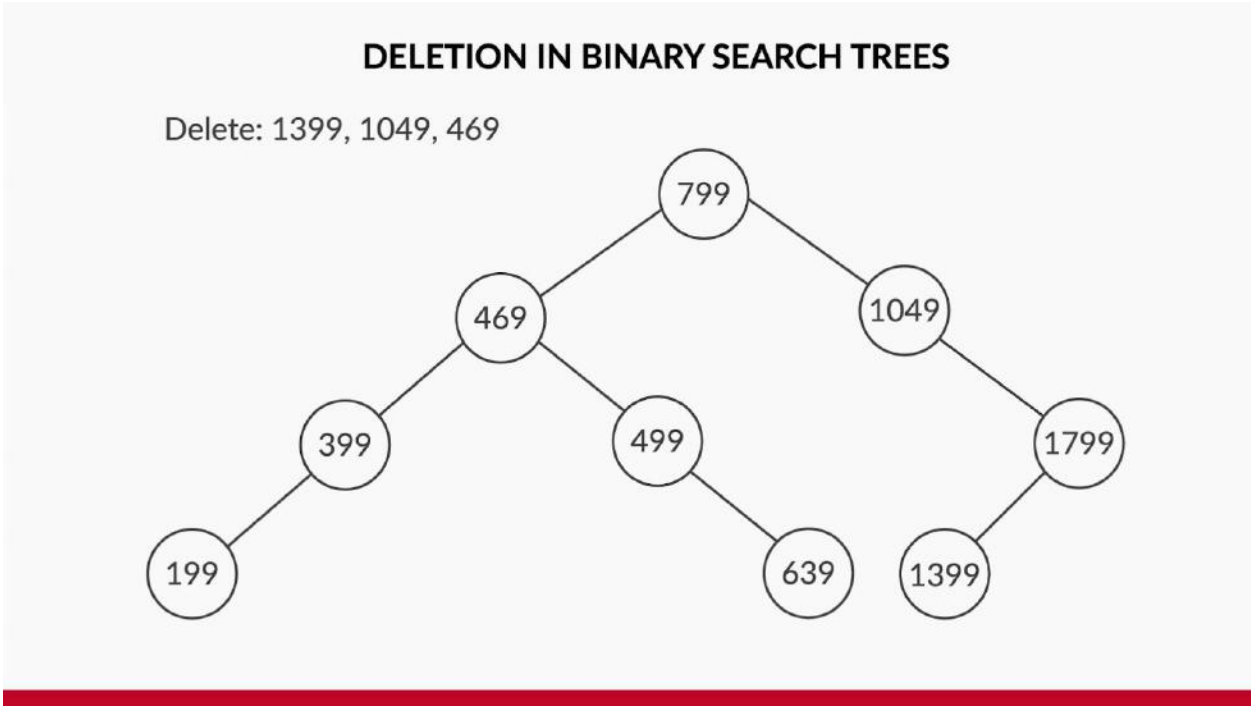


Figure 9: Deletion in binary search trees

Deletion of data from binary search trees can be carried out in three ways:

1. If the node to be deleted is a leaf node, it is removed from the tree without making any alterations to other nodes
2. If the node has a child, then it is deleted, and its child node is connected to the parent node
3. If the node has two children, you need to find its successor to replace it with. The successor node would be the minimum node in the right subtree or the maximum node in the left subtree.

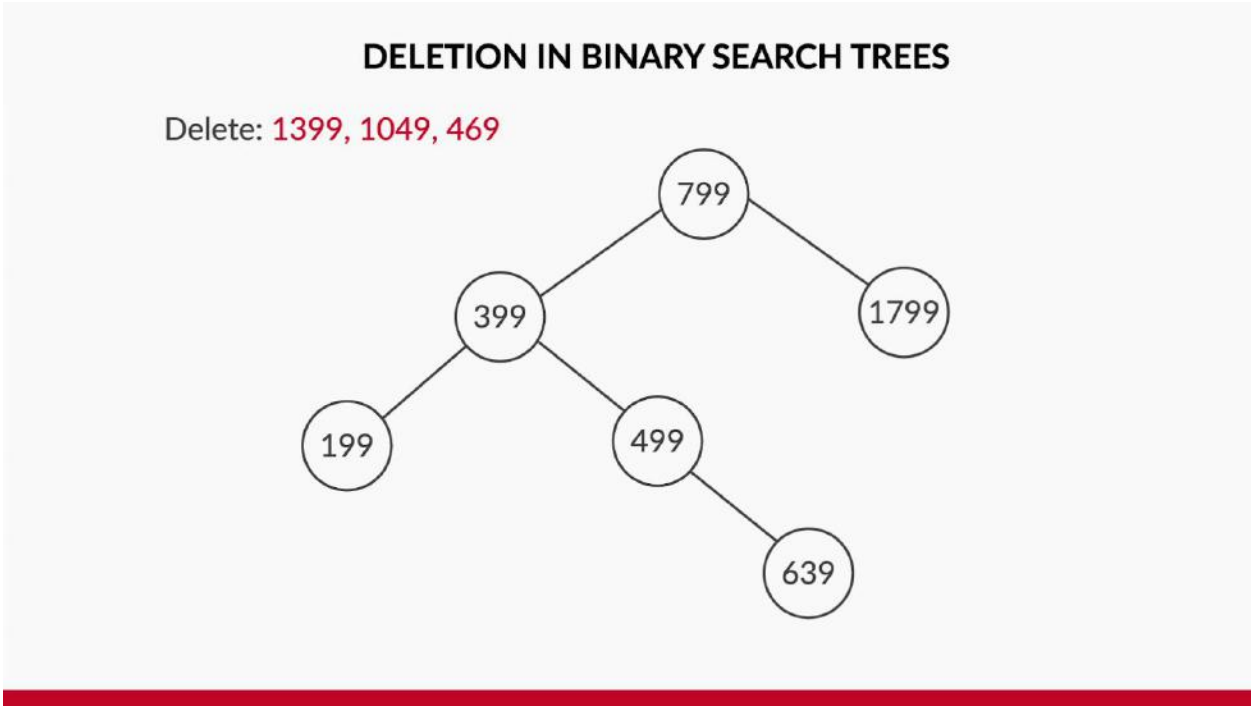


Figure 10: Deletion in binary search trees

Note: The drawback with the deletion of data is that the tree becomes slightly unbalanced and this condition can be handled by using self-balancing binary search trees such as AVL trees, red-black trees, etc.

Data structure	Time complexity					
	Insert		Delete		Search	
	Average case	Worst case	Average case	Worst case	Average case	Worst case
Sorted array	O(n)	O(n)	O(n)	O(n)	O(Log(n))	O(Log(n))
Balanced binary search tree	O(Log(n))	O(Log(n))	O(Log(n))	O(Log(n))	O(Log(n))	O(Log(n))

K-Dimensional Trees

In range queries, if the range involves only one constraint/variable, then a binary search tree can be used for range queries. Usually, however, the number of constraints will be more than 1. To handle such queries, one needs to use advanced tree data structures like K-Dimensional trees (also known as KD trees). The increasing complexity and volume of the data are what leads to the need of using K-Dimensional trees to segment it.

The K-Dimensional tree is an extension of the binary search tree that partitions spaces to organise multidimensional data points in a K-Dimensional space. This helps to store and find data efficiently when it lies within some range of the dimensions (or attributes) that describe it.

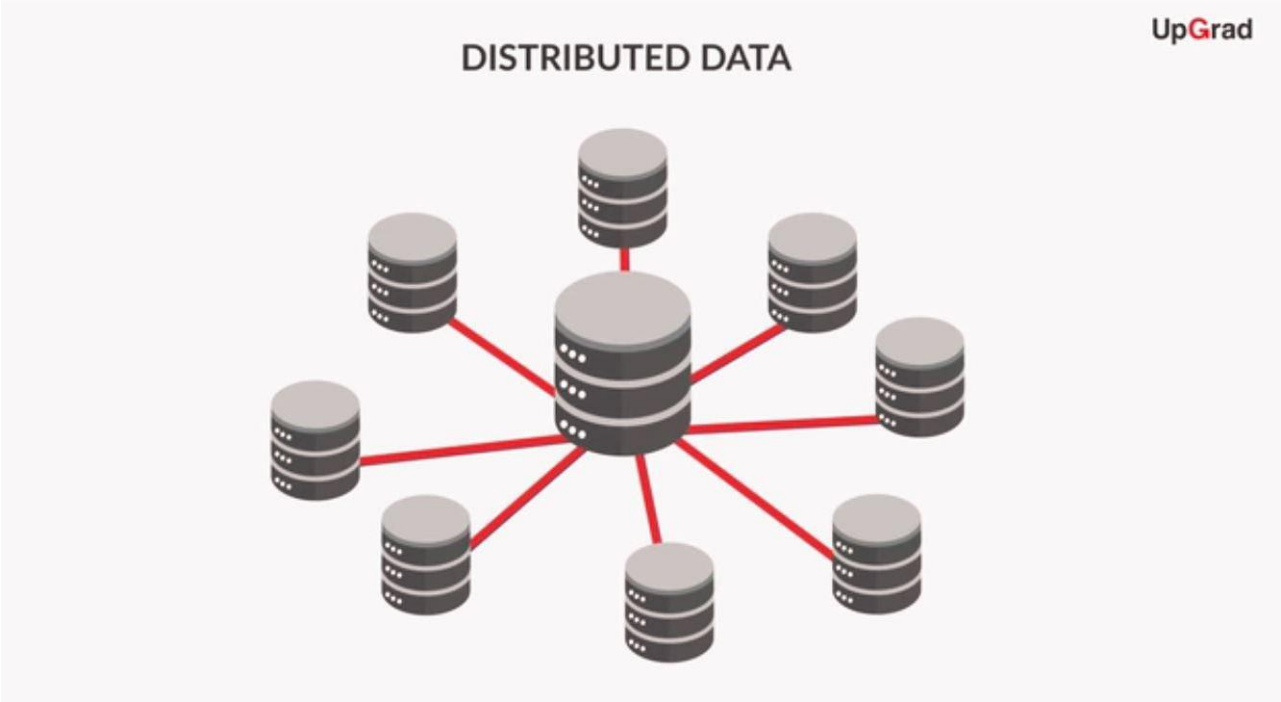


Figure 11: Distributed data

Note: The 'K' in KD tree stands for the number of dimensions of the data that is stored in the tree.

Construction of KD Trees

There are various methods to construct a KD Tree. One of the methods is as follows:

- Recursively partition the points along the data axes alternatively
- Then, build a two-dimensional tree by storing the resulting partitions in the nodes of the tree

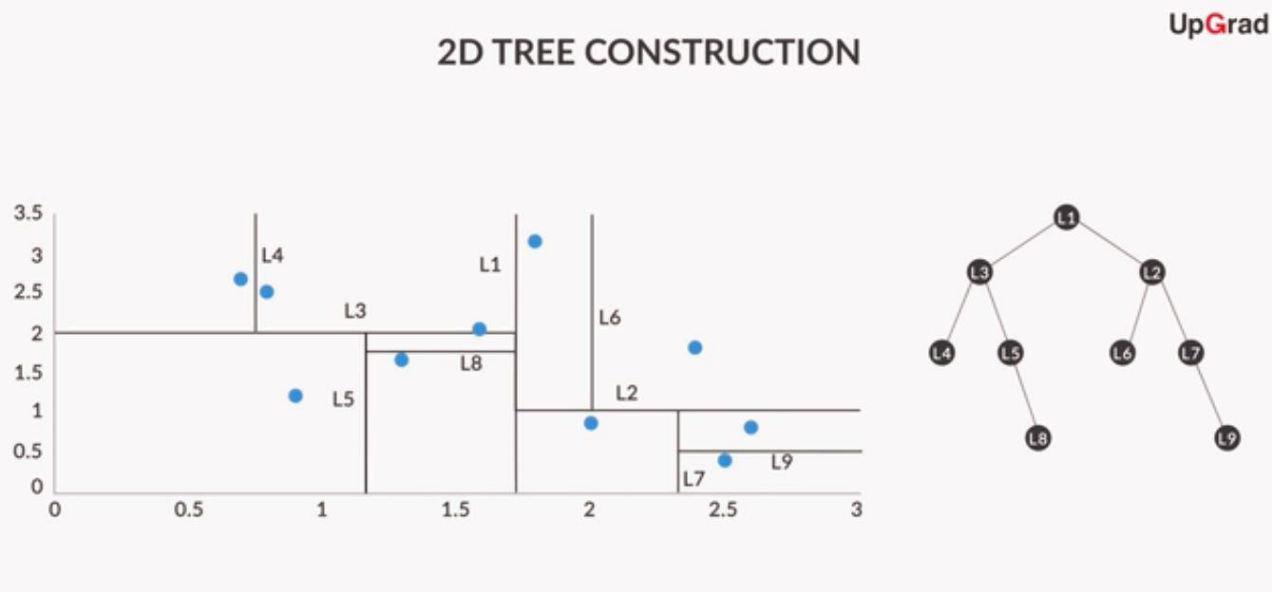


Figure 12: 2D Tree construction

Range Queries in KD Trees

Range queries in K-Dimensional trees are similar to those in binary search trees. The only new condition one needs to check is: when comparing a node, one needs to compare only the coordinates of the partitioning dimensions of that node.

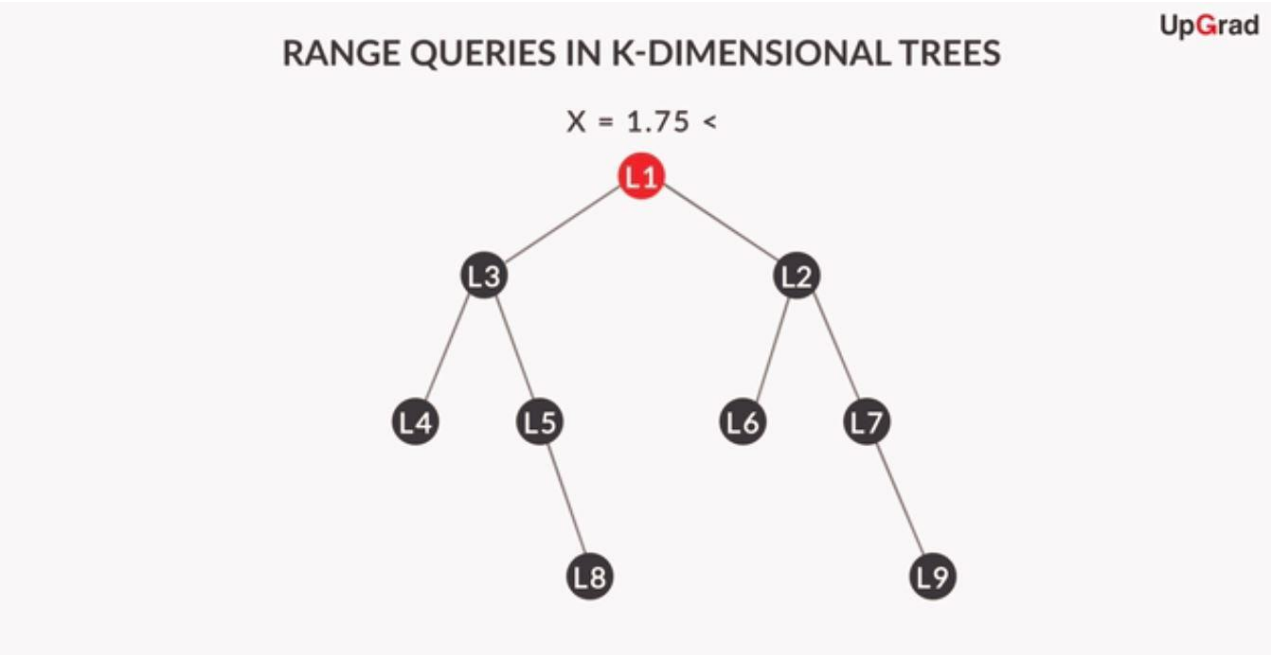


Figure 13: Range queries in K-Dimensional trees

Applications of KD Trees

These days, we use various apps on mobile phones, from gaming apps to apps for shopping, etc. In several of these apps, we often see ads displayed on the screen. For example, in gaming apps, ads are displayed as banners or videos in between game levels. These ads are handled by ad networks.

An ad network is an exchange that connects advertisers looking to show ads with apps willing to host these ads. When you open an app on your handheld device, it sends out a request to the ad servers used by the ad network, communicating that it is ready to receive and serve ads. Upon receiving this request, the ad network serves the most relevant and profitable ads in its repository, which is built through collecting deals from advertisers. To decide on the relevance and profitability, the ad network needs to garner information from the device or apps sending the ad request. For this purpose, the ad request contains several data facts, such as the device ID, app ID, IP of the device, timestamp, latitude, longitude, number of ad units, requested price, etc. The facts contained in the ad request cannot always be used in their raw format as they are to make intelligent decisions. They must be translated or transformed into other forms first.

One of the most important facts used for mobile ad-relevant systems is the location of the device. When an app sends an ad request, it supplies the latitude and longitude of the device obtained from the system's GPS. However, as stated before, the raw latitude and longitude data won't be very insightful. They must be translated into geographical administrative units such as addresses and ZIP codes to be useful. Administrative entities such as ZIP codes are associated with third-party data such as demographics, weather, average income, and language. This is very useful to ad relevance engines. A KD tree may be used to implement this translation from latitude and longitude to ZIP codes. The two dimensions of the tree will be latitude on the y-axis and longitude on the x-axis. The latitude and longitude of the geographical centroid of each ZIP code are used to represent a node of the KD tree.

Note: The main advantage of a tree is eliminating data points from a large search space. Hence, one should attempt to make a tree as balanced as possible so that, on each lookup, he/she may eliminate half of the search space.

There are different variants of K-Dimensional trees just as there are many different ways to construct them. There's another method of constructing much-balanced KD trees, which is as follows:

1) The points to be inserted are sorted according to their coordinates on the axis that partitions the space.
2) The median of the points to be inserted is selected, and it is made the root of the tree or subtree.
3) Similarly, the points to the left and right sides of the median are recursively inserted into the respective left and right subtrees.

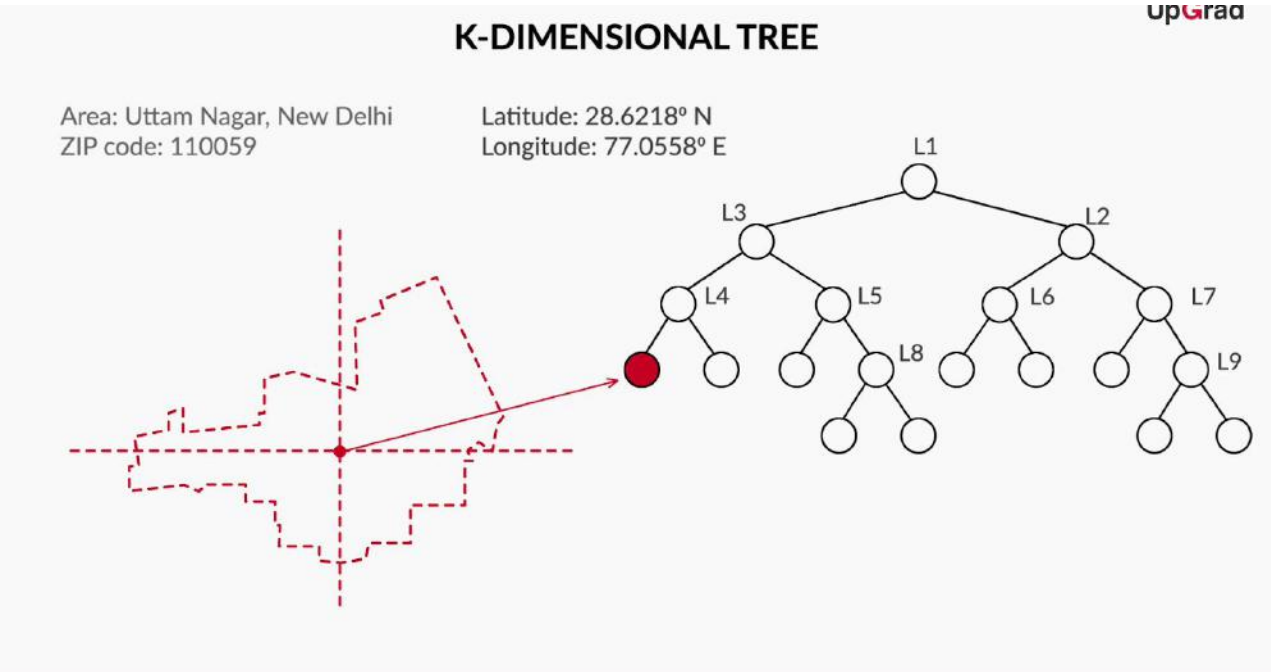


Figure 14: K-Dimensional tree

Once the tree is built, the steps involved in locating an incoming ad request to its nearest ZIP code are as follows (nearest neighbour detection):

1. Starting from the root node, you need to keep traversing the tree, alternating between the latitude and longitude, until you reach the leaf node (i.e. you move to the left or right node depending on whether the search point of the ad request is less than or greater than the current node in the split dimension).
2. Once you reach the leaf node, you save it as the 'current best'.
3. For the nearest neighbour detection to be complete, the algorithm needs to unwind the recursion on the tree. While unwinding —

a. If the current node is closer than the current best, it becomes the current best itself (closer is determined by the squared distances between the search node, the current best, and the current node).
b. The algorithm also checks the nearer points on the other side of the splitting point.

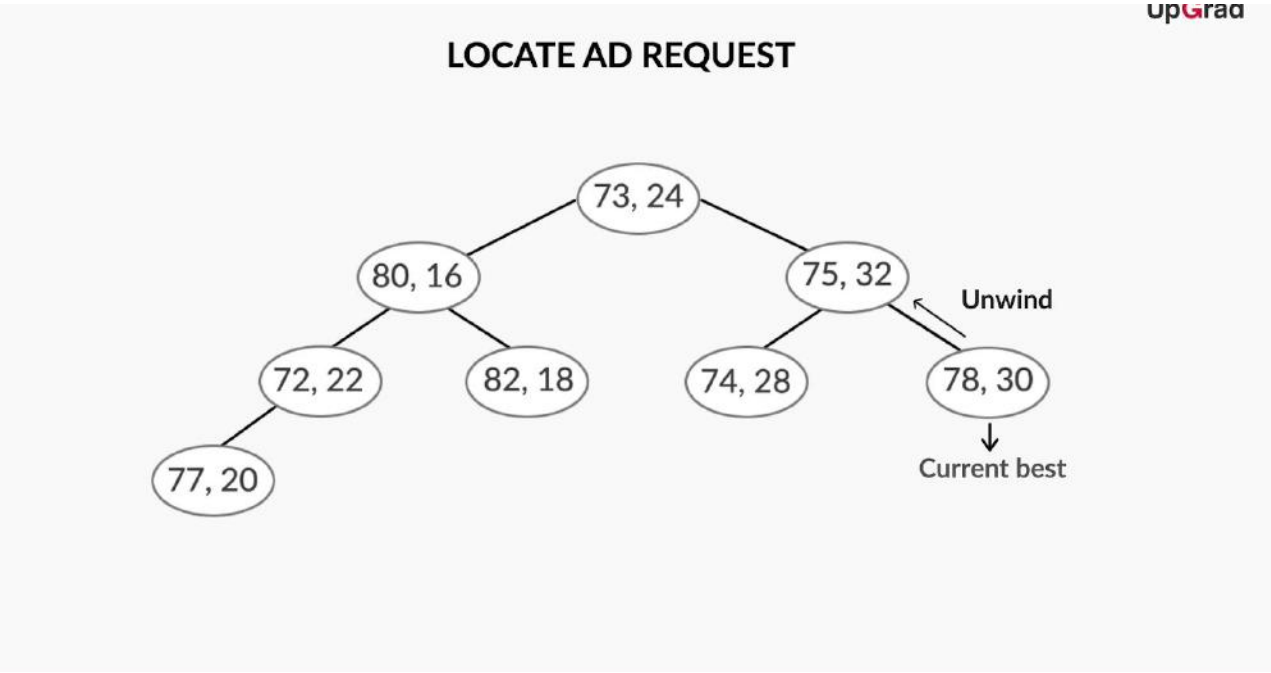


Figure 15: Locating ad request

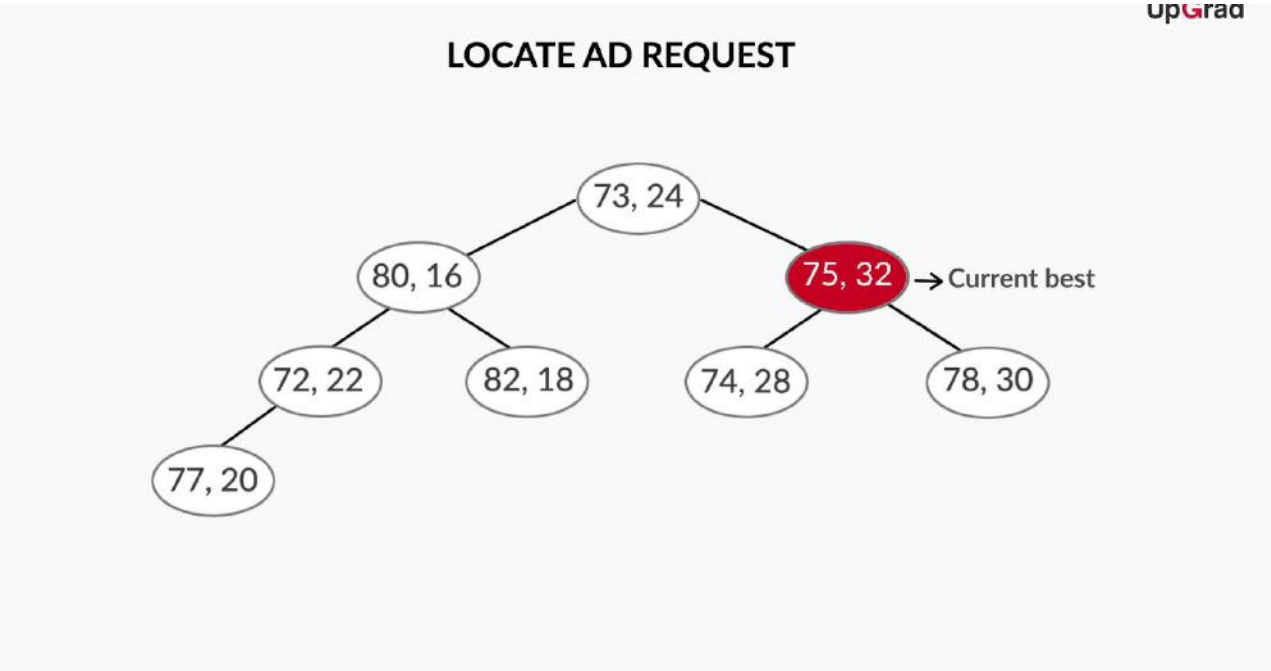


Figure 16: Locating ad request

Note:

1. Sorting all the nodes at each stage of construction takes a long time, especially when the number of nodes is really huge.
2. In practical scenarios, for large spaces, only a random subset of values is picked and sorted to select a median. This practice may sometimes lead to slightly unbalanced trees, but these are worthy enough to compensate for the huge pre-processing cost of sorting.

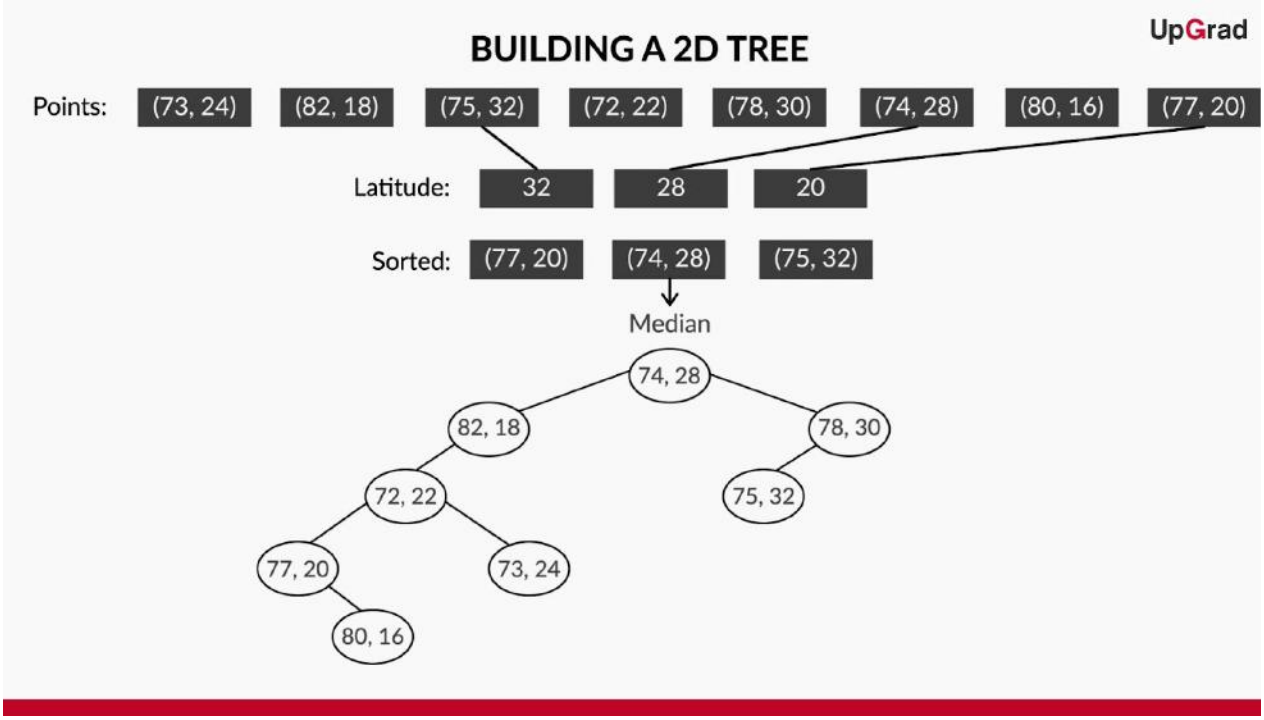


Figure 16: Building a 2D tree

Lecture Notes

Virtual Machines and Amazon EC2

This module dealt with the concept of 'virtualisation' — a technique that allows a computer to perform the tasks of multiple computers by sharing its hardware resources across multiple environments. The second session of the module introduced you to cloud computing and the popular cloud service provider 'Amazon Web Services' (AWS).

What is Virtualisation?

Virtualisation is a technique that allows you to use the full capacity of a physical machine by distributing its capabilities among multiple environments. It provides an abstract environment through virtual hardware, a software environment, an operating system, a storage device, or network resources, to run a wide variety of applications.

To understand the concept of virtualisation, you considered an example of server consolidation. Server consolidation is an approach to efficiently using all the resources of a computer server to reduce the total number of servers or server locations available. This can be done by running the servers on a single piece of hardware, possibly a multicore machine with slightly high-end computing resources.

In the example you saw, an organisation had four physical servers with individual dedicated purposes, as you can see in the image given below.

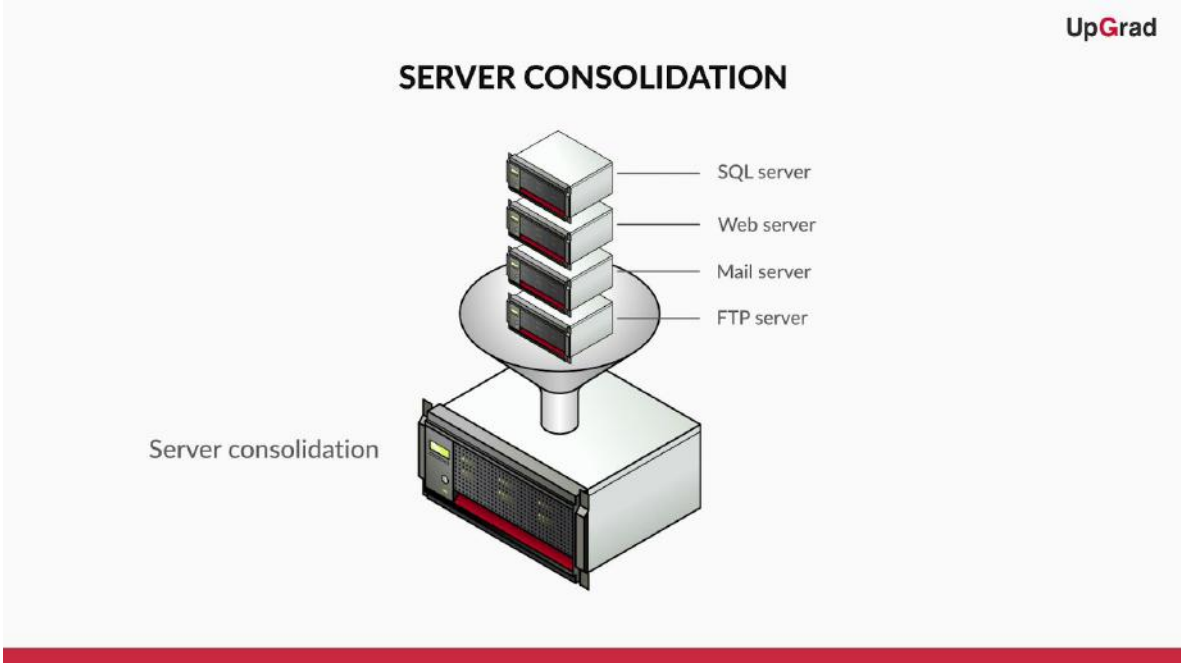


Figure 1: Server Consolidation

One was an SQL server for storing user information, and the other three were a web server for hosting websites, a mail server for emails, and an FTP server for sharing files. All the four servers were scattered across the globe, across different continents. The organisation wanted to consolidate them instead of having four different physical servers, as given in the image below. This method of using a virtualised environment increases hardware efficiency and allows you to run each server on a single machine.

Traditional vs Virtual Architecture

In a traditional machine, as you can see in the image below, there is a hardware layer (which is a combination of a central processing unit, physical memory, and I/O devices). Above this layer, there is an operating system, and above the operating system, there is an application layer.

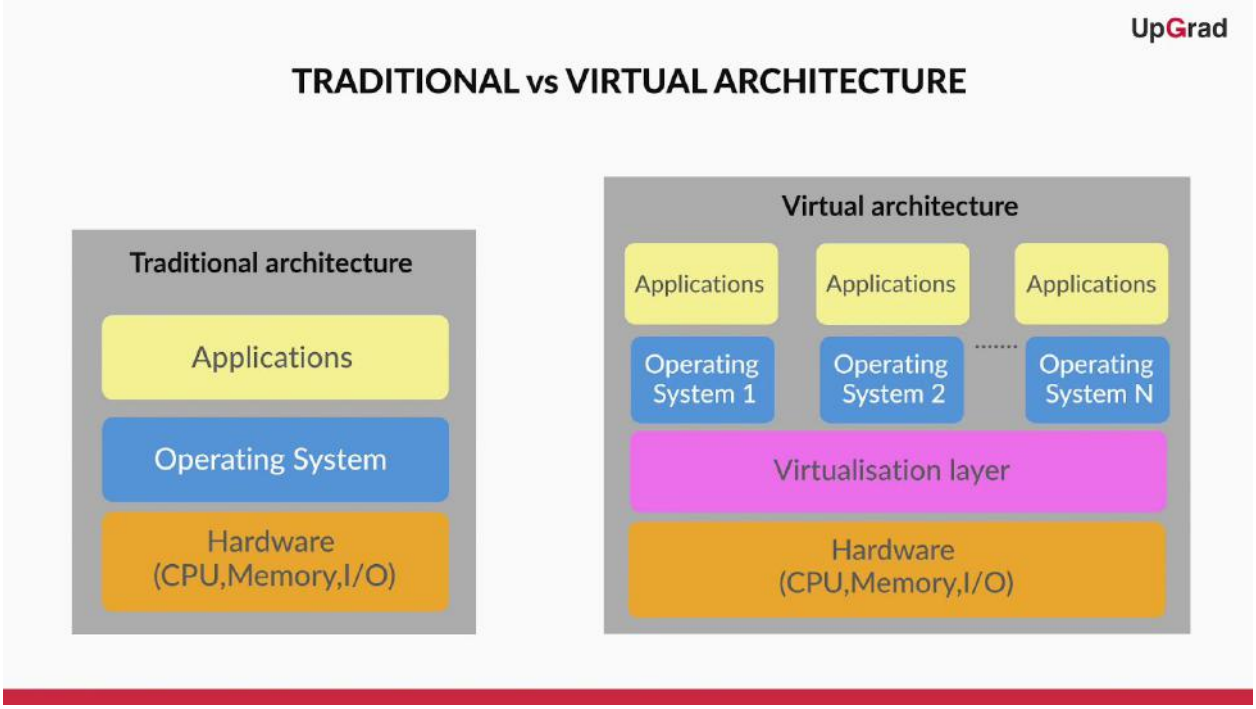


Figure 2: Traditional vs Virtual Architecture

In a virtual machine, there is a virtualisation layer, which could be a piece of computer software, firmware, or hardware, that creates and runs virtual machines. A computer on which a virtualisation layer runs one or more virtual machines is called a host machine, and each virtual machine is called a guest machine.

Advantages of Virtualisation

Virtualisation helps to optimise the usage of underutilised hardware and software resources. In a non-virtualised environment, you may have a set of hardware and software resources that are not utilised to their full potentials.

For example, consider a multicore machine that is running a few applications that use only one core; this machine's other cores are being underutilised. Now consider the example shown in the image below:

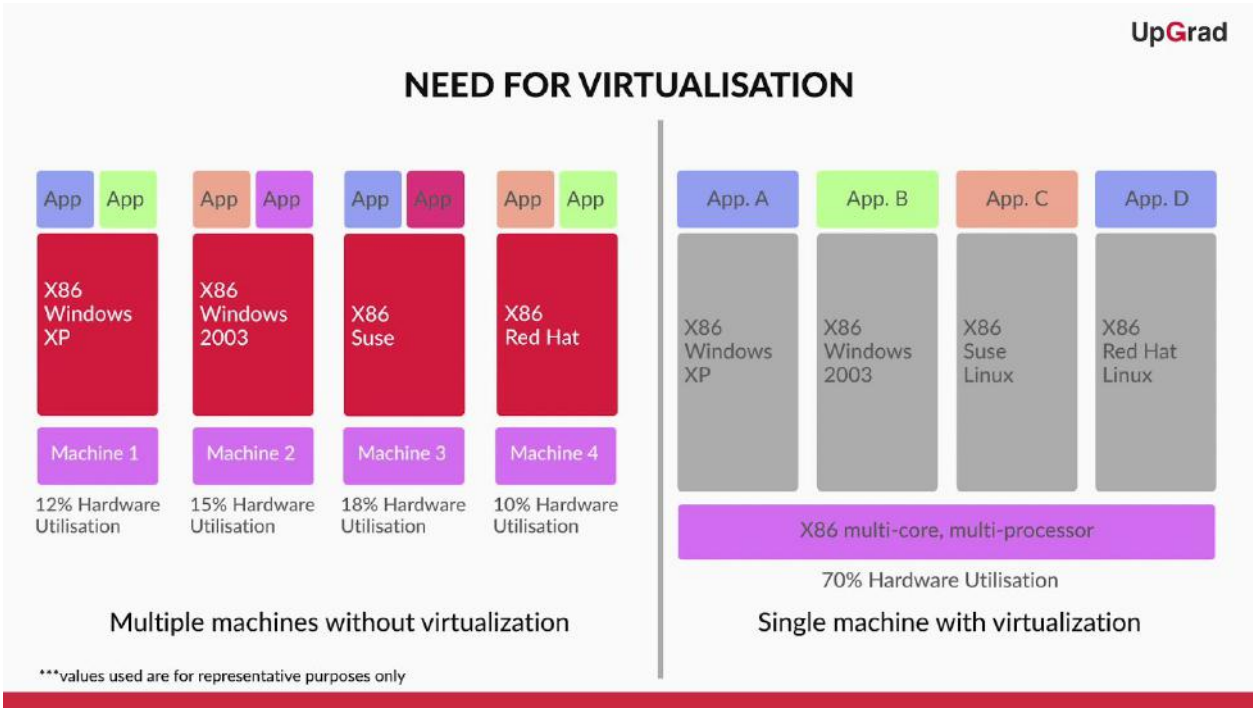


Figure 3: Need for Virtualisation

On the left-hand side, there are several different architecture types without virtualisation — X86 running Windows XP, X86 running Windows 2003, Linux, Red Hat, etc. In addition, there is another set of applications that are running on those operating systems. The percentage of hardware utilisation in each case is 12, 15, 18, and 10, respectively, which gives a combined hardware utilisation of 55%.

However, in a virtualised environment, as shown on the right-hand side of the image, the hardware utilisation improves drastically and goes up to 70%.

Thus, in a virtualised environment, you can reduce the costs and manage the resources more efficiently. This is one of the benefits or advantages of running such an environment — it improves the performance of the hardware.

Hypervisor

The virtualisation layer contains a piece of software called the 'hypervisor', which dynamically shares the hardware resources. The hypervisor is also known as a 'virtual machine monitor'. It isolates the operating systems and applications from the underlying hardware.

A hypervisor controls the hardware on which there are several guests and VM operating systems that provide different execution environments. So, a virtual machine is an isolated run-time environment or an emulated computer system capable of running software programs or a guest operating system that

exhibits the behaviour of a separate computer. Usually known as a guest, a VM is created within another computing environment that's referred to as a host.

Multiple virtual machines can exist within a single host at one time, where the hypervisor controls the host processor — that is the CPU — and other resources, allocating what is needed to each virtual machine, in turn, and ensuring that they do not disrupt each other.

There are two types of hypervisors available. They are —
1) Bare-metal or native hypervisors, and
2) Hosted hypervisors

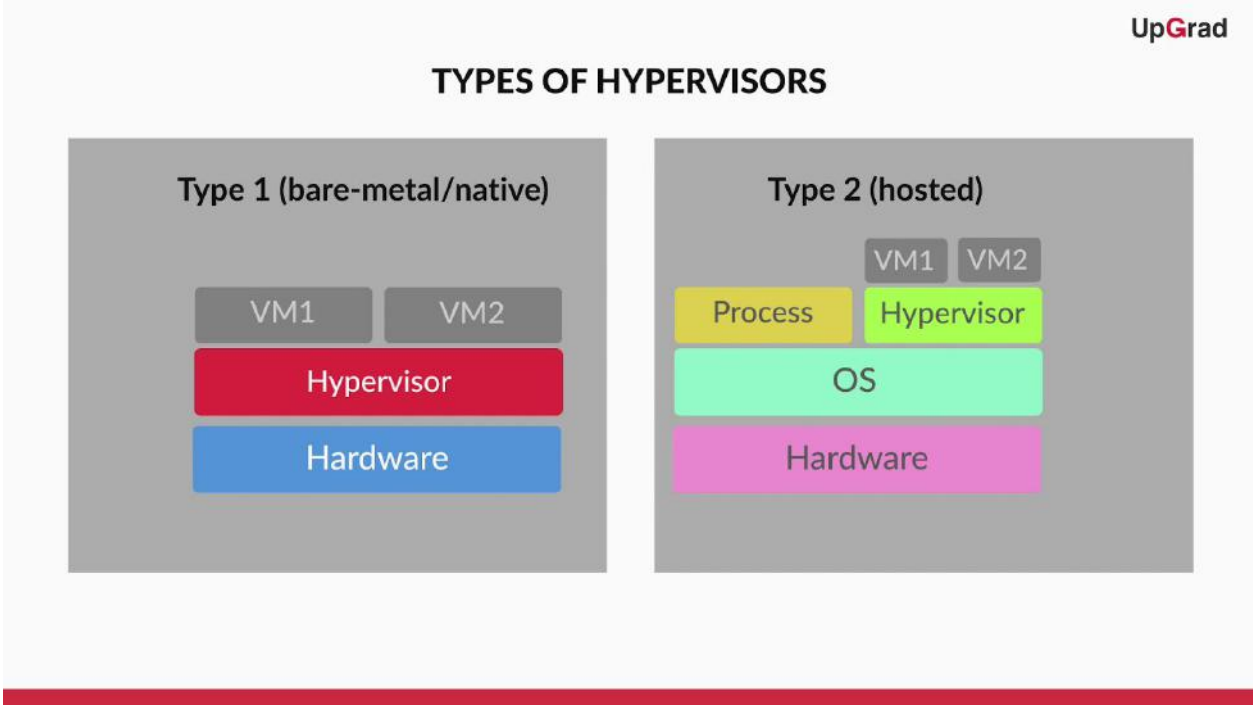


Figure 4: Types of Hypervisors

A **bare-metal hypervisor** sits above the hardware and has full control over it. Since it resides directly above the hardware, it is called a 'bare-metal' hypervisor.

Some examples of bare-metal hypervisors are VMware ESX, Microsoft Hyper-V, Xen, etc.

In the case of a **hosted hypervisor**, a conventional operating system sits in between the hypervisor and the hardware. This hypervisor abstracts guest operating systems from host operating systems. Guest operating systems need a share of the hardware to carry out their functions; this hardware is completely controlled by the operating system that sits above it. The hypervisor interacts with this operating system and ensures that the functionalities required by the guest operating systems are satisfied or are provided by the hardware through the operating system.

Some examples of hosted hypervisors are VMware Workstation, Microsoft Virtual PC, Oracle VirtualBox, KVM, etc.

What is Cloud Computing?

Cloud computing is one of the most overused buzzwords in the tech industry. It is used as an umbrella term for a wide array of platforms, services, and systems. Virtualisation is often confused with cloud computing. Although the two technologies are similar, the terms are not interchangeable. Hence, it is not entirely surprising that there's a great deal of confusion regarding what they mean exactly.

Virtualisation is achieved by software that separates physical infrastructure to create various dedicated resources. Virtualisation software makes it possible to run multiple operating systems and multiple applications on the same server at the same time. Such software enables businesses to reduce IT costs while increasing the efficiency, utilisation, and flexibility of their existing computer hardware. Hence, virtualisation makes servers, workstations, storage devices, and other systems independent of the physical hardware layer.

Cloud Computing	Virtualisation
1. It refers to a service that results from virtualisation.	1. It is the foundational element of cloud computing and helps deliver its value.
2. It is the delivery of shared computing resources, software, or data — as a service and on demand — through the internet.	2. It is a piece of technology that makes up cloud computing.
	3. It can be termed as the keystone of cloud computing.

Apart from shared independent access to a computing resource, cloud computing also provides the following services:

a) On-demand self-service: It empowers the user to assign computing resources to him/herself whenever required.
b) Broad network access: It provides access to multiple devices on a network.
c) Rapid elasticity: It allows the user to add or reduce capacity through software.
d) Measured service: It keeps track of who is using what and how much.

Why is Cloud Computing Important?

Big data applications involve processing vast volumes of data. With the increasing demand for processing data in real time, there is a need for a hardware system that has an immense computing power. The growing popularity of cloud computing and cloud data stores has encouraged organisations to use the same for big data storage and processing.

Before the existence of cloud-based systems, whenever an organisation had to set up its own data centre, the ideal way to do so was to set up a physical data centre with all the required hardware, etc. In other words, the organisation that set up the centre was the sole owner of it. Hence, it was responsible for the maintenance of the data centre to ensure that it was always up and running.

Fortunately, things have changed — based on the magnitude of the problem being solved, an organisation can rent the adequate amount of resources. Storing and processing big data require proper hardware resources, which may sometimes become costly and hard to maintain. With cloud providers, anyone can enjoy the computational power of, say, a supercomputer by paying just a few dollars. Hence, by using cloud-based storage and processing systems, an organisation saves itself from all the struggles of setting up and maintaining its own computational and storage hardware.

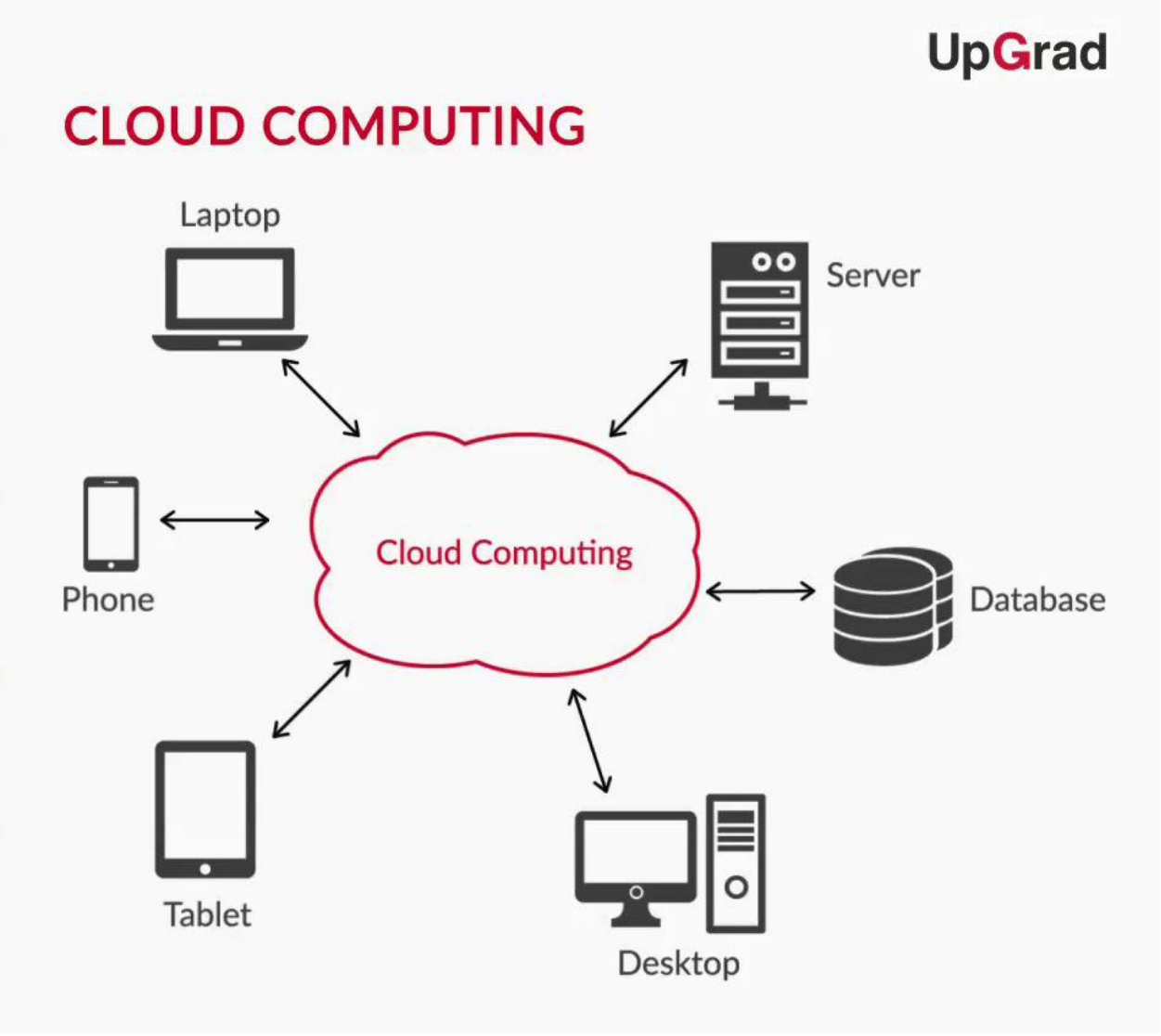


Figure 5: Cloud Computing

Apart from being cost-efficient and provisioning resources on demand, some other benefits of using cloud-based systems for storing and processing big data are —

1. **Agility:** Traditional systems are difficult to scale. To accommodate exponentially growing data, cloud systems provide access to thousands of virtual servers. These servers work together to accomplish complex processing tasks seamlessly, in short spans of time.

2. **Feasibility and convenience**: Traditional systems were not easily scalable. They needed additional physical machines to increase their processing and storage powers. However, because of the virtual nature of cloud systems, the user now has access to an unlimited amount of resources that can be used whenever needed. With the cloud, depending on the nature of the problem, enterprises have the flexibility to choose the desired amount of processing power and storage space easily and quickly.

Amazon EC2

Amazon Web Services (AWS) provides you with managed services on its cloud platform. It manages software installation and hardware maintenance and provides the necessary resources to users for the running of big data systems, thereby freeing them from the burden of managing any hardware, software, network, etc. Users may then perform a one-time set-up process, beyond which, everything is managed by AWS. Some popular services provided by Amazon Web Services are as follows:

1. Amazon Elastic Compute Cloud (EC2): Compute service
2. Amazon Simple Storage Service (S3): Storage service
3. Amazon Relational Database Service (RDS): Database service
4. Amazon Virtual Private Cloud or VPC (VPN): Network service
5. Amazon Simple Notification Service (SNS): Application integration service

Amazon EC2 of AWS provides computing resources, with complete control over them. These resources are both secure and scalable. Hence, there is an added advantage that you need to pay only for the resources that you use.

Algorithm Design Lecture Notes

Previously, you were introduced you to a variety of data structures, which are used to store and manipulate data. If you recollect, you utilised these data structures to address the user's concerns efficiently.

In other words, you were asked to solve the user's problem, and you selected the most efficient data structure to implement a solution to the problem.In this module, you learnt about the techniques to obtain solutions to problems i.e. we introduced you to an algorithm design technique that is particularly relevant in the big data context.

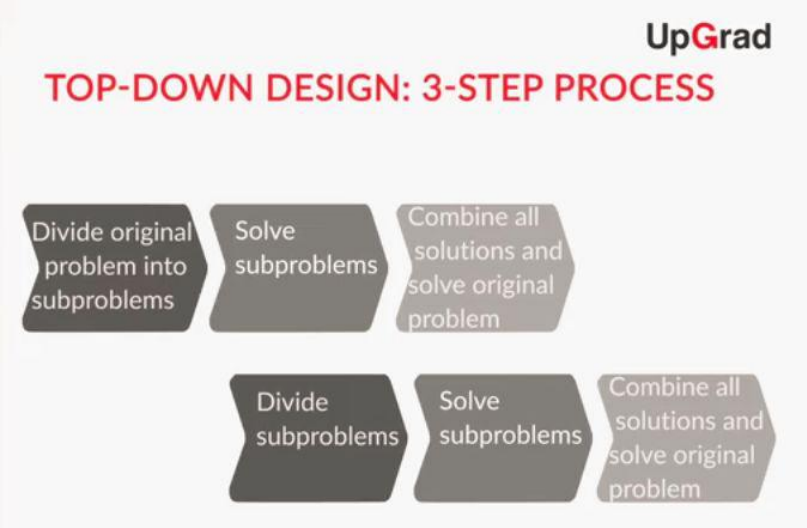
Top-down Approach to Algorithm Design

Top-down design is an approach where you start with a problem, divide the problem into subproblems and find solutions to those subproblems. This is essential to decrease the complexity of the problem. As the problem is complex, you break it down into smaller problems and solve those subproblems.

Top-down design is a three step process where you-

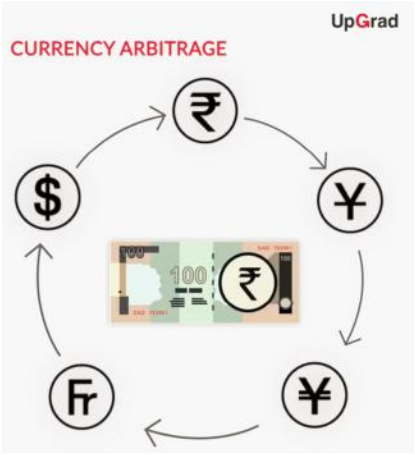
1. Break the original problem into subproblems
2. Solve each of those subproblems
3. Combine the solutions to get a solution to the original problem.

The way to solve these subproblems is: you apply the same steps mentioned above for each subproblem.

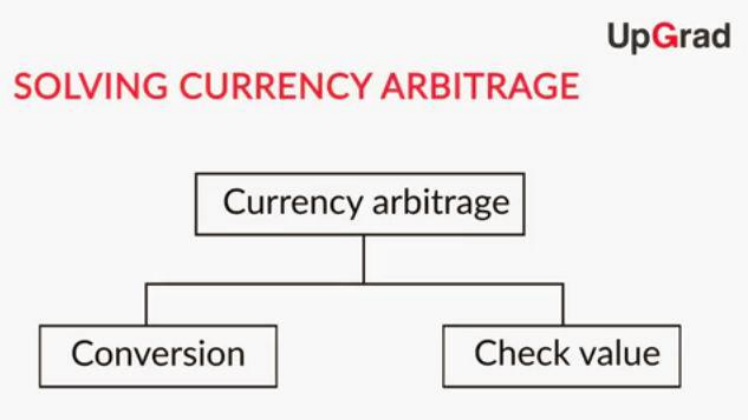


Top-down design is a recursive approach and the process stops when you encounter atomic problems or problems for which solutions exist. Atomic problems are defined as problems for which solutions are either trivial or there already exist solutions, therefore you need not further subdivide the problem into subproblems.

Consider the problem of currency arbitrage where multiple currencies are being traded in the market concurrently and you are looking for a profit by clicking a currency and converting it to a sequence of other currencies and then seeing whether you can retrieve a larger value.



For instance, if you take 100 rupees, convert it to some number of Yen and convert those Yen into Yuan and then to Francs, then Francs to Dollars and finally convert those Dollars back to Rupees, you may find that you ended up with more than 100 Rupees which you started with.

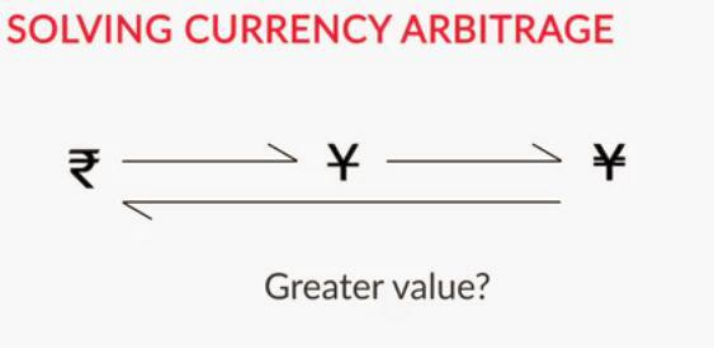


If you wish to solve this problem of checking whether there exists a sequence of currency exchanges or currency conversions that leads to a profit, you can break this problem into two subproblems as follows:

1. Conversion: Conversion of one currency to another. If you know how to solve this subproblem then you can repeat the process i.e. the combination. You can repeat the process from one currency to another currency.

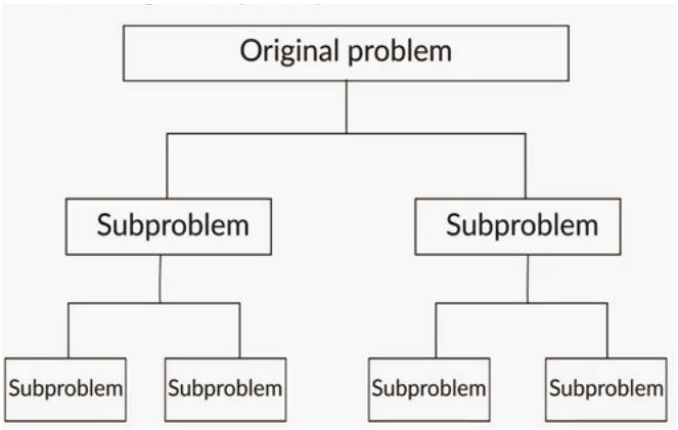
2. Check Value: Checking whether getting back to the original currency will yield the larger value than what you started with. You perform this step after every conversion.

Now once you have identified these two subproblems, their combination will give you the solution to the original currency arbitrage problem. In this case, the solution will be to repeatedly convert a currency to a newer currency and check whether the currency you have can be converted to the first currency to get a value that is larger than the original value.



Despite its advantages Top-Down design suffers due to a few limitations:

- The first step of dividing the problem into subproblems is an acceptable division only if you know how to combine the solutions. 'An acceptable division of problems and subproblems is one where you know how to combine the solutions to the subproblems to get the solution to the original problem.'



- The next challenge arises in deciding the number of subproblems i.e. in deciding how many subproblems a problem has to be broken down into. If you keep dividing a problem or subproblem further down, you many end up with a large number of subproblems, making it difficult to manage the complexity and carry out the computation. To manage this complexity, always divide the problem into small number of subproblems.

To summarize, in the top-down approach, you divide a given problem into subproblems and combine them later to obtain the solution to the original problem. The same approach is taken to solve each sub-problem wherever possible until atomic problems are encountered. The solutions to these atomic problems are either known or computed directly. The number of subproblems is kept small to manage the complexity of the overall algorithm.

Divide and Conquer

In this segment, we talked about divide and conquer which is an algorithm design technique that is a special case of top-down design.

In divide and conquer as in top-down design, we divide the problem into subproblems but one or more of the subproblems will have the same structure as the original problem. That means solving the original problem and solving the subproblem is one and the same task.

One major limitation of Top-Down design is:

- If you keep dividing problems further into subproblems, then each subproblem may have a different structure. Then, every time you divide, you need to look for new solutions as well.

This brings us to divide and conquer design, wherein

- You divide the problems into subproblems that have the same structure as the original problem. So in this case the process of dividing and combining the solutions again are similar at every level. Once you know how to combine the solutions for the subproblems, then you can recursively follow the same procedure to obtain the solution for the original problem. The process of combining at all levels is going to be the same because you are dividing the same kind of subproblems.

TOP-DOWN VS DIVIDE AND CONQUER

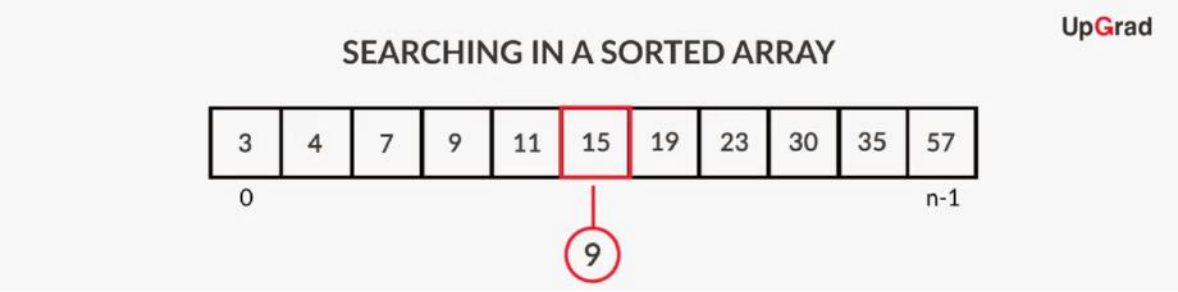
Top-Down	Divide and Conquer
Subproblems may have different structures	Every subproblem has the same structure as the original problem
New solutions are required	Dividing and combining are similar at each level

- So essentially in divide and conquer: You need to look for subproblems that have the same structure as the original problem. Hence, divide and conquer is a special case of top-down design, where one or more of the subproblems have the structure same as that of the original problem but with a smaller size. This enables us to repeat the same

division process and the same combination process again and again to obtain an algorithmic solution.

Binary Search Algorithm (DIvide and Conquer)

Consider the problem of searching in a sorted list i.e. you have to find a value k in a list going from index 0 to index (n-1). Assume that the list is sorted in increasing order.



Here you can look at this list say if a value that you are looking for is found in the left half of the sublist then the value must be less than or equal to the median value. If the value is to be found on the right half of the list then the value should be greater than the median in the list.
Therefore you can convert this problem into two subproblems:

- Searching in the left sublist
- Searching in the right sublist

The combination process is simple if you look up the median value and if It turns out to be the value that you are looking for, then you are done. If the value that you are looking for is less than the median value, then you will search in the left sublist i.e. go from index 0 to mid-1 where mid is the index of the median value. Similarly, you will perform the same on the right sub list i.e. going from mid+1 to n-1 if your value is more the median.

UpGrad

BINARY SEARCH ALGORITHM

```
BINARY-SEARCH (LS, lo, hi, k)
 1. if lo>hi
 2.     return -1
 3. mid = floor((lo+hi)/2)
 4. if x == LS[mid]
 5.     return mid
 6. If x<LS[mid]
 7.     BINARY-SEARCH(LS, lo, mid-1, k)
        //lo=0 for the initial array
 8. if x>LS[mid]
 9.     BINARY-SEARCH(LS, mid+1, hi, k)
        //hi=n-1 for the initial array
```

This is a classic example of divide and conquer process because searching in the sublist is the same problem as searching in the original list. The size has changed and has come down from n to n/2 approximately depending on the parity of n. But the problem of searching is still the same to the right or left sublist that is sorted.

In summary, divide and conquer is a very useful algorithm design technique and consists of three stages:

- Divide a given problem into subproblems with the same structure as the original problem.
- Conquer the subproblems by solving them recursively. If an atomic subproblem is obtained, solve it directly.
- Combine the solutions to the subproblems to obtain the solution to the original problem.

Efficient Divide and Conquer Techniques

In divide and conquer, we divide the problem into subproblems where all the subproblems share the same structure with the original problem.

It is always possible that your division is possible in multiple ways i.e. for the same problem, there are multiple ways to divide the same problem into subproblems, therefore yielding multiple divide and conquer solutions or multiple divide and conquer algorithms for the same problem.

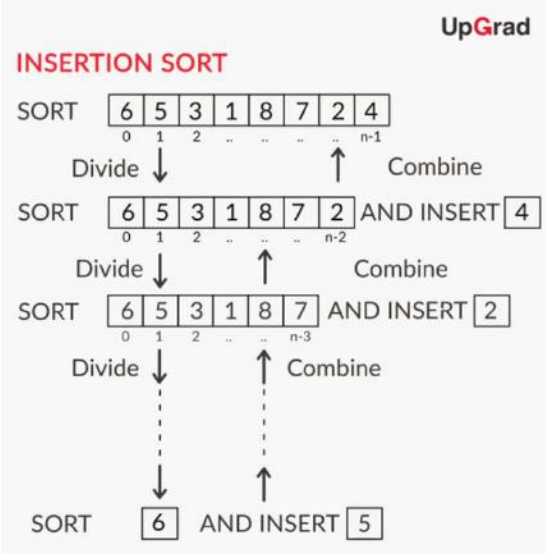
Insertion Sort

You are given a list of size n, indices ranging from 0 to n-1. Now in order to sort this list you can look at this problem and break it down into a subproblem .

- First sort the list of size n-1 i.e. the sub-list LS with indices going from 0 to n-2. Now the combination will be inserting one particular value i.e at the last position LS[n-1] into the sorted list.

- So you started the problem of sorting a list of size n and now there is a subproblem which is sorting the list of size (n -1) and the solution can be used to construct the solution to the original problem by inserting the last element into a sorted list.

- Now if you look at this subproblem of sorting a list of size (n-1), you can break this subproblem into another subproblem, where you can sort a list of size (n-2) and into which you can insert the (n-1)th element.



So this has the same structure as the original problem and hence the same combination will work. Therefore the problem of sorting is solved by repeatedly inserting one element at its correct position into a sorted list. This will terminate when the size of the sublist becomes 1.

UpGrad

INSERTION SORT ALGORITHM

IN-SORT (LS, n)	INSERT(LS, n, k)
1. if n≤1	1. i = n-1
2. return	2. while i≥0 and LS[i]>k
3. if n>1	3. LS[i+1]=LS[i]
4. IN-SORT(LS, n-1)	4. i = i-1
5. INSERT(LS, n-1, LS[n-1])	5. LS[i+1] = k

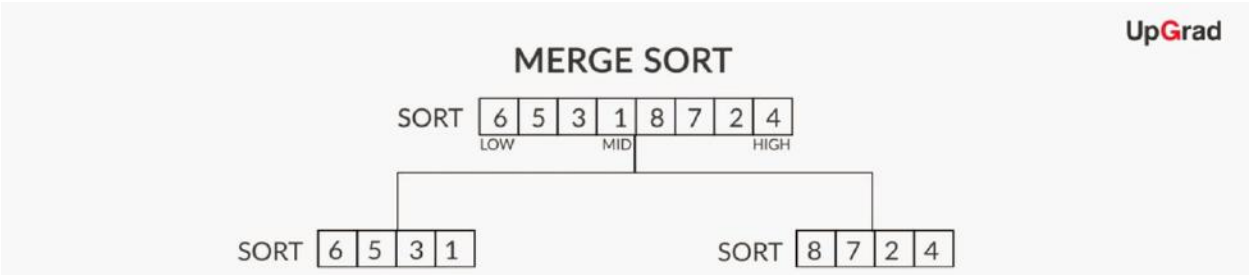
This naturally leads to an algorithm called IN-SORT that takes a list LS and size n. If the size is less than or equal to 1, you get a return because that is already a sorted list. If the size is more than 1, then you call IN-SORT(LS, n-1) followed by the INSERT function. So, IN-SORT(LS, n -1) sorts the list with n- 1 elements. Having done that, you insert into the sorted list the last element LS[n- 1] to obtain the final sorted list.

Merge Sort

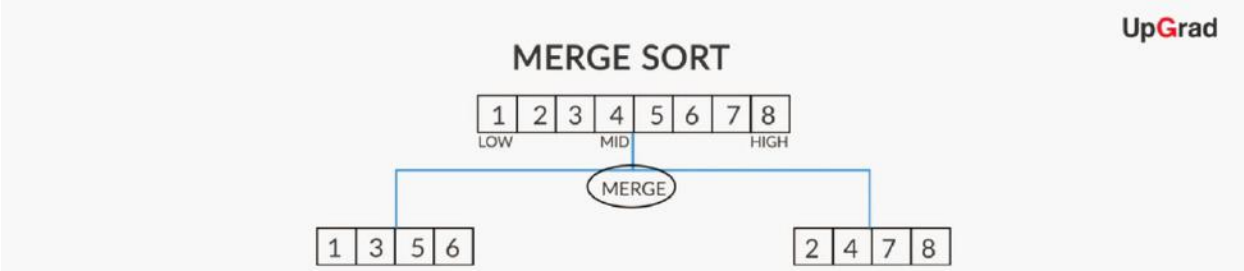
We discussed how to solve the same problem of sorting in a different way but again using the divide and conquer technique. Let's recall the merge sort algorithm.
If you have to sort the elements of a list in ascending order, you can divide the list into two halves or two sublists and then sort them separately. So you have two subproblems:

- Sorting the left sublist which goes from low to mid where mid is the index of the middle element
- Sorting the sublist on the right which is LS with indices going from (mid+1) to high.

UpGrad



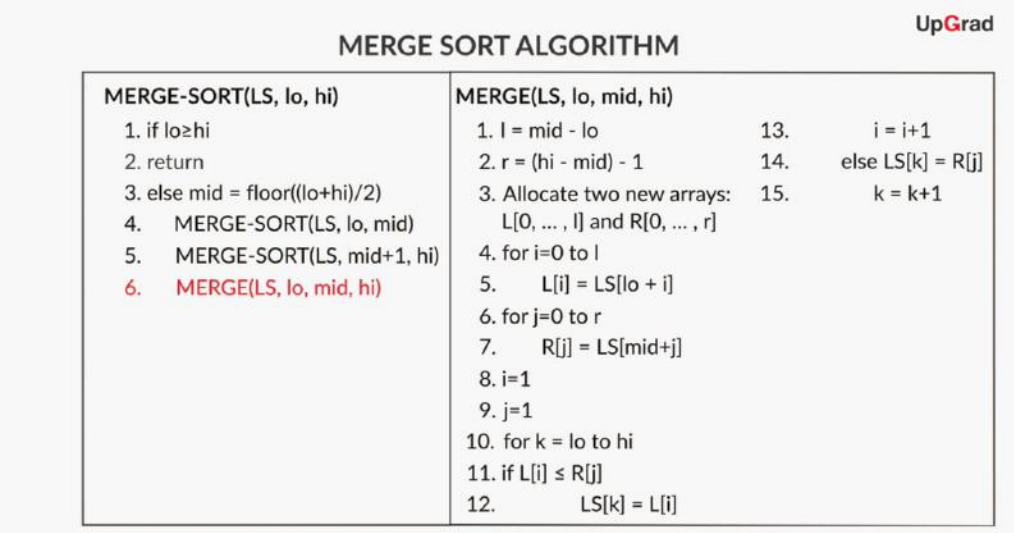
Assume that you have these two sublists sorted. Then you need a procedure that merges two sorted lists into a longer sorted list. Here, the division step divides the list into two sublists of roughly equal sizes, and the combination step consists of merging the sorted sublist. So merging the sorted sublist is a problem that you need to solve.



Indeed, the process of dividing the problem into two subproblems can be repeated because the structure of these problems is the same as that of the original one, i.e. sorting a list.
To sort one of these sublists, you can divide this sublist into two sublists until the size of the sublist becomes 0 or 1. When the size of the sublist is 0 or 1 then you can say it has already been sorted. Then you can stop the division and start combining from there on.

You can convert this process into an algorithm:

- Takes a list LS and two indices lo index and hi index as input.
- Check whether the lo index is higher or lower than hi index or rather equal to hi index, where if it is equal you don't have anything to do because it is a list of size 0 or size 1.
- If the size of the list is greater than 1, you can compute the index mid=floor((lo + hi)/ 2). Then you can invoke merge sort on the left sublist which goes from index lo to index mid and you can separately call sort on right sub list which goes from index mid+1 to index hi.
- Finally, you call a procedure named merge in which you take two sublists and merge them into a single sorted list.



This is also a recursive procedure because merge is calling itself and that's expected since you started with the original problem of sorting, divided it into two subproblems each of sorting a sublist.

Comparing the running time of Merge Sort and Insertion Sort:

INSERTION SORT	MERGE SORT
O(n^2)	O(n log n)

In divide and conquer process, if your division yields subproblems of roughly equal size then your algorithm would have better efficiency.
So in summary:
Reducing the size by a constant factor at every stage yields efficient divide and conquer algorithms. Examples include-

- Merge sort reduces the size by half at every stage and has a worst-case running time of O(nlogn). While insertion sort reduces the size by 1 at every stage and has a running time of O(n^2).

A more elementary example involves searching in sorted arrays:

- Binary search discards half of the array at every stage and has a running time of O(logn) while primitive/sequential search discards only one element at a time and has a running time of O(n).

If your algorithm reduces the problem size to a fraction *a* of its original size at every stage, then it will finish dividing in $(log_{1/a}\ n)$ steps, e.g. a=½ for binary search and hence it has $log_2\ n$ steps of division in the worst case. The challenge is to divide and combine the subproblems at every stage efficiently. If the total work of division and combination requires O(1) time, as in binary search, the overall running time will be O $(log_{1/a}\ n)$.

If the total work of division and combination requires O(n) time at every stage, as in merge sort and quicksort, the overall running time will be O(n $log_{1/a}\ n$).

Limitations of Divide and Conquer

The technique of Divide and Conquer has a limitation where if you are unable to find a subproblem that has the same structure as the original one, then you cannot apply this technique. So in this case, one has to fall back to top-down design in general.

The limitations of Divide and Conquer are as follows:

- The first step in seeking a divide and conquer solution is to see whether there exists a subproblem that has a structure similar to that of the original problem. If such a subproblem doesn't exist, you need to fall back on top-down design.

- The second limitation of divide and conquer is that sometimes the subproblems you identify are exactly the same, not just the same structure but they also have the same size. In short, the technique may yield exactly same subproblems at different levels. The subproblems cannot have the same size as the original problem because the subproblems have sizes which are smaller than the original problem. But if the two subproblems at two different levels originating at two different subproblem levels have the same size and the same structure and you don't recognise this then you will be repeating computations i.e. the solution will be inefficient.

A simple illustration of this limitation is a divide and conquer algorithm for computing Fibonacci numbers. Fibonacci numbers are defined by the recursive relation:

$F(n)=F(n-1)+F(n-2)$,
$F(2)=F(1)=1$

The recursion yields:

$F(n)=F(n-1)+F(n-2)$ in the first stage and

$F(n-1)=F(n-2)+F(n-3)$ in the second stage.

Note that $F(n-2)$ is computed twice in this algorithm when $F(n)$ is called. Similarly, every Fibonacci number $F(k)$, k lesser than n, is computed multiple times. This results in additional overhead and increases the running time of the algorithm.

In Summary:

- Divide and conquer technique works for a problem only when there exists a method to divide the problem into subproblems sharing its structure, i.e. the subproblems are smaller instances of the original problem.

- Further, the performance of a divide and conquer algorithm is drastically affected if there exist subproblems with the same size at different levels.

Linear Data Structures Lecture Notes

This module deals with the Abstract Data Type(ADT) Dictionary and its implementation using elementary data structures like arrays(unsorted and sorted) and linked lists. The second and third sessions introduce advanced data structures, namely hash tables and Bloom filters.

Introduction

Most real-life applications require storing a list of records and retrieving them later. The Dictionary or Map ADT is designed to facilitate this requirement. In a dictionary, each record or entry is identified using a key. Hence, these keys are required to be unique. For instance, in a phone book application to store and look up contacts using names, each contact is a record and its name is the key.

A dictionary supports three fundamental operations: ADD, FIND and DELETE.

1. The ADD operation takes a record as input and inserts it into the dictionary.
2. The FIND operation takes a key as input and outputs the corresponding record. If no such record exists, it outputs null.
3. The DELETE operation takes a key as input, returns the corresponding record and deletes the same. If no such record exists, it returns null.

Implementation using Elementary Data Structures

A dictionary may be implemented using various data structures. The choice of data structure is governed by its implications on the performance of a given application. The following table captures the running time of the ADD and FIND operations for arrays and linked lists.

	ADD	**FIND**
Unsorted Array	O(1)	O(n)
Sorted Array	O(n)	O(log n)
Linked list	O(1)	O(n)

1. Linked lists are preferred for applications where the total number of records cannot be estimated, e.g. a social networking database.
2. Arrays are preferred for applications with information about the total number of records.
    A) An unsorted array is used if the relative number of ADD operation calls is very high due to O(1) running time of the ADD operation.
    B) A sorted array is used if the relative number of FIND operation calls is very high due to O(log n) running time of the binary search operation.

Note: A dictionary is generally dynamic, i.e. records are added, retrieved and even deleted frequently. In special cases, a static dictionary may be required where all records are added at once and only the FIND operation is called subsequently. A sorted array is the preferred data structure in such instances.

Preprocessing Cost and Amortised Cost

Consider a dictionary in which all records have been added and sorted by key. If a large number of FIND operations are to be performed on this dictionary, then each of these has a running time of O(log n). However, the binary search operation with O(log n) running time may be used here only if the array has been preprocessed, i.e. sorted beforehand. Hence, a preprocessing cost needs to be associated to the running time of the FIND operation here.

At the same time it's unjustified to associate the preprocessing cost with every single FIND operation in a sorted array. Is there a number that can be associated with the expected running time of the FIND operation in a sorted array? This leads to the topic of amortized cost.

The **amortized cost** per operation for a sequence of operations is defined as the total cost of these operations divided by the number of operations.

- In our case, let us suppose we sort our map using one of the best algorithms you are familiar with: mergesort. We know that this method runs in O(n log n) time.

- In the sorted map case, sorting would be done in n log n time and each FIND operation would be carried out in O(log n) time using binary search. So, the amortized cost per operation for `m' search operations, where m >> n would be O(n log n + m log n)/m = O(log n) for our sorted map. On the other hand, the amortized cost for an unsorted map would be O(mn)/m = O(n) since each search operation takes linear time. This clearly shows the time optimization obtained by using sorted maps.

- For a sorted map, if m(search operations) is equal to one, then the amortized cost will be $\frac{n \log n + m \log n}{m} = (n+1)\log n \simeq O(n \log n)$. However, if m is close to n, then the amortized cost is $\frac{n \log n + m \log n}{m} = \frac{2n \log n}{n} \simeq O(\log n)$.

The preprocessing cost i.e. the cost to sort the array depends on the sorting algorithm. For instance

- Insertion sort is used in practice for small lists, when the list is small or the size of the list is small, then insertion sort works better than other algorithms.

- Quicksort is fairly efficient in practice; its expected time complexity is O(n log n), whereas it also gives the worst case of O(n^2). Quicksort is often the most commonly used algorithm in practice. If the list size is large, then quicksort performs better than insertion sort. Please note that both of these are used for in-memory sorting.

- The other most commonly used algorithm is merge sort. Merge sort is typically used when the data is in secondary memory, such as tape or disk or if the data is stored in a file. Since it's easier to access it sequentially, merge sort is preferred. Merge sort actually gives you O(n log n) algorithm in the worst case as well as average case, but the constant factor(the order complexity) for merge sort is higher than that of quicksort.

  Quicksort running time = $k_1 n \log n$ **,**
  Merge sort running time = $k_2 n \log n$ **,** $k_1 < k_2$

- Typically when the data is stored in secondary memory such as a tape or a disk or sometimes transferred over a network, you make use of merge sort and use quicksort for sorting the items present in primary memory.

	WORST CASE RUNNING TIME	AVERAGE RUNNING TIME	USE CASES(where the algorithm performs the best)
INSERTION SORT	$O(n^2)$	$O(n^2)$	Small lists stored in primary memory
QUICKSORT	$O(n^2)$	$O(n \log n)$	Large lists stored in primary memory
MERGE SORT	$O(n \log n)$	$O(n \log n)$	Lists stored in secondary memory and networks

The Bin sort Algorithm and Direct Addressing

You know how to sort the elements in an array based on comparison of keys and the minimum order for performing such a sort is O(n log n). But you can perform a sort without comparing keys using the Bin sort Algorithm.

The bin sort procedure or algorithm works for a list of records whose keys are unique and numeric and lie in a known range. It sorts them by indexing, for example, if there are 900 records whose keys lie in the range [100,...,999], then the record corresponding to each key K is indexed as (K−100), i.e. stored in an array of size 900 at the index (K−100).

For performing a Bin sort you need to make the following assumptions:

- The keys are numeric
- The keys are unique
- The keys are in a particular range that is known.

CODE FOR BIN SORTING is as follows:

```
for(i= 0; i< n; i++) {
lsnew[ls[i].key - low] = ls[i]
}
```

Here **ls** is a list of n records with unique keys and **low** is the lowest key in the list of n records, **lsnew** is the new sorted array obtained after bin sort. For every i^{th} record, you subtract the lowest key from its key value to obtain an index, say inew and then place the i^{th} record in index inew of the new array **lsnew**.

The time complexity of bin sort is **O(n)**, where n is the size of the list, since there will be **n** unique iterations for the **n** records present in the list. To **FIND** a certain record in the sorted list, you simply need to provide the key of the record and thus the result of performing bin sort on a list of records is that the FIND operation runs in constant time.

Direct addressing is a powerful method to implement a dictionary. Indeed, the fundamental operations run in constant time when direct addressing is used. But the major drawback of this method is its memory requirement and the associated sparsity i.e. direct addressing requires huge amount of memory, which could at times be impractical and virtually impossible. This paves the way to the next fundamental data structure Hash tables, which primarily solves these problems.

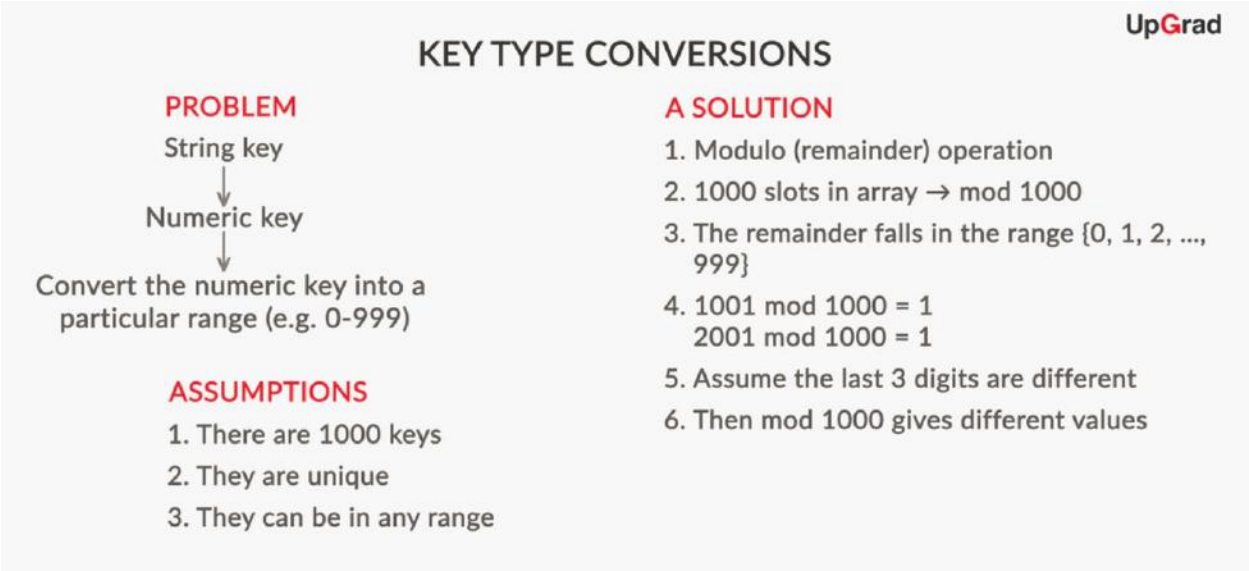
Hash Table Introduction

If your keys are in a known range and, they're numbers then you can map them to integer locations or indices of the array locations. If your keys are in a known range but they are not numeric then you can find a way to map any type of key to a numeric key and then restrict the range so that you will end up with locations or array indices and then use the same approach that you did in Bin sorting. For this process, you need to make use of Hash functions.

For instance, consider a range of strings which are known. You can map the string key to a numeric key using the hash function. Hash function also solves the issue faced with direct addressing, i.e. its huge memory requirement and the associated sparsity.

Hash Functions

A hash function is defined as a function that takes keys of any specified type as input and outputs numbers, corresponding to array indices. A hash function initially converts a given key to a possibly large integer and then reduces it to a smaller number, typically using the modulo operator. For a hash table of size n, the mod n operator is used to obtain indices less than n. These array indices are referred to as hash values. Each record is then stored in the hash value corresponding to its key.



Non-numeric keys such as strings are initially converted to integers(possibly large) using a hash code. A simple example of a hash code is to add the ASCII values of the individual characters of a string. The integer so obtained is then reduced to the range of available indices using a modulo operator.

Suppose you have 1000 records where one record has a string key “Hello”. To convert this string key to a numerical key you can make use of a Hash code. Assume this string to be an array of characters, then you can convert all the characters to the lowercase and add the individual ASCII values of each character. So “Hello” will give a sum of 372. To convert this integer to the corresponding numerical key you can make use of a modulo (Remainder) in this case. So this is a simple formula to convert any string of characters to a number once you have done this conversion you can do the modulo operation to get a value in the location range 0 to 999 so modulo operation will depend on what is the **range of numbers** or the number of locations. The mod value depends on the size of the table. If the table size is 1000, then you

need to do modulo thousand and if you have 2,000 values in your array, then the location range will go from 0 to 1999, therefore you will do modulo 2000. So for a table size of 1000 your hash will be 372.

This process is commonly referred as **Hashing**, where you convert any key to a desired numeric key. You need to keep in mind that each key must be unique or it may result in collision or overwriting of the 2 or more records at the same location.

Similarly you can take each value in the original list, compute a hash function which gives you a location and you can put that value or drop the value into that table in that location. This procedure is in line with the process of Bin Sort, where bin sort uses a simple subtraction operation instead of a hash operation.

You can find a particular value with the same efficiency you achieved in Bin sort. One of the results of doing Bin sort was that you could locate a particular element by simply obtaining the location in constant amount of time and therefore the cost of find operation was constant. Similarly, you can take the key and do a hash operation here and locate the index to insert or read a record. So you locate the index in constant time and hence, the effect is the same as that in Bin Sort.

Hash table Design and Performance Analysis

Hash functions enable adding and retrieving records in constant running time while requiring lesser memory than direct addressing. Let's introduce you to a data structure that is designed using hash functions.

By now you know that a HASH TABLE is an array of n locations. The indices in a Hash Table go from on to n-1. You need to assume that your hash function gives unique values in the range of {0, 1 ,..., n-1} and your input key can be any arbitrary value.

So let's assume that you have an array H with indices from 0 to (n-1). You have a hash function which you assume returns a location in this range 0 to (n-1) for any key K where K could be string type for example. So with this assumption, you can write your add function. The add function essentially takes a record R and R has a key and that key can be hashed. The hash function will give a location as an output for storing the record in the Hash Table.

UpGrad

THE ADD OPERATION

H[0... n-1]

hash(K)

ADD(R)

1. i = hash(R.key)
2. H[i] = R

RUNNING TIME

1. Constant running time
2. Two operations
   a. Compute the hash function to get the index
   b. Use the index to store the record

Hash Function offers a constant running time for the ADD operation as it requires 2 operations, each running in constant time-

a. Computing the Hash Function to get the index
b. Use the index to store the record

Similarly you can do a FIND operation. The find operation takes the key of a record and returns a value of the location where the record can be found in the Hash Table.

UpGrad

THE FIND OPERATION

FIND(K)

1. i = hash(K)
2. return H[i]

RUNNING TIME

1. Constant time operation
   a. Compute the hash to get an index
   b. Return record

Hash Function offers a constant running time for FIND operation as it requires 2 operations, each running in constant time-

a. Computing the Hash Function to get the index
b. Returning the record from the index

Note: Hashing the key and hashing the record mean the same things.

For a given application, a hash function should :

- Accept any possible key as input and
- Output one of the indices(locations) in the allocated array (i.e. table).

The latter i.e. requirement is under the control of the designer/implementer: i.e. one can ensure this requirement by choosing the size of the array (table) to be equal to the modulus of the hash function.

For a good Hash function you should always keep the following points in mind:

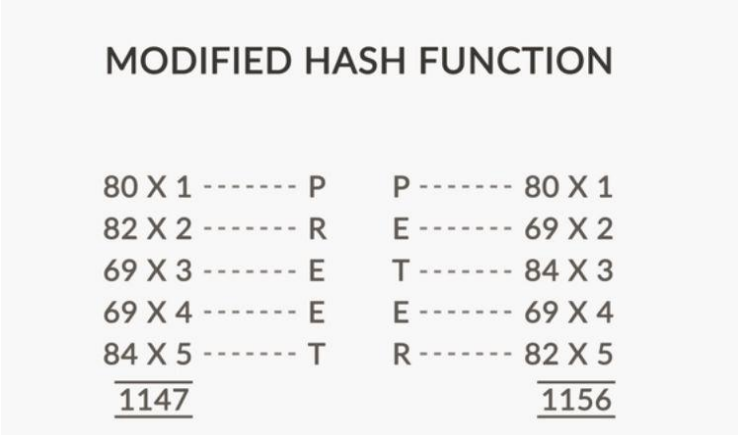
- "Use" all the information in the input (i.e. in the keys).
- Distribute the keys uniformly across the array indices.
- Output different hash values for keys that are "similar yet unequal".

If a Hash function allots all the records to only a set of hash table indices instead of uniformly distributing them or fails to make use of all the information provided by the record key, then such a hash function is usually unacceptable and is designated as a bad Hash function.

Collision and Chaining

For the ADD and FIND operations in the previous segment, we made an assumption that the keys will be mapped to unique locations. But that assumption is not always true. For instance, for the hash function that we designed, we took the ASCII values of the characters and added them up. If 2 strings have similar (PETER & PREET) names, then such a hash function will fail as the sum of the ASCII values of characters will be the same for both strings.

The records will then get stored in the same location which means one will overwrite the other. The phenomenon of two distinct keys K1 and K2 getting hashed to the same index is known as collision in hash tables. That is, K1 ≠ K2 but h(K1) = h(K2).
Collision cannot be eliminated, but may be reduced to a great extent by designing good hash functions. One method is shown in the figure here

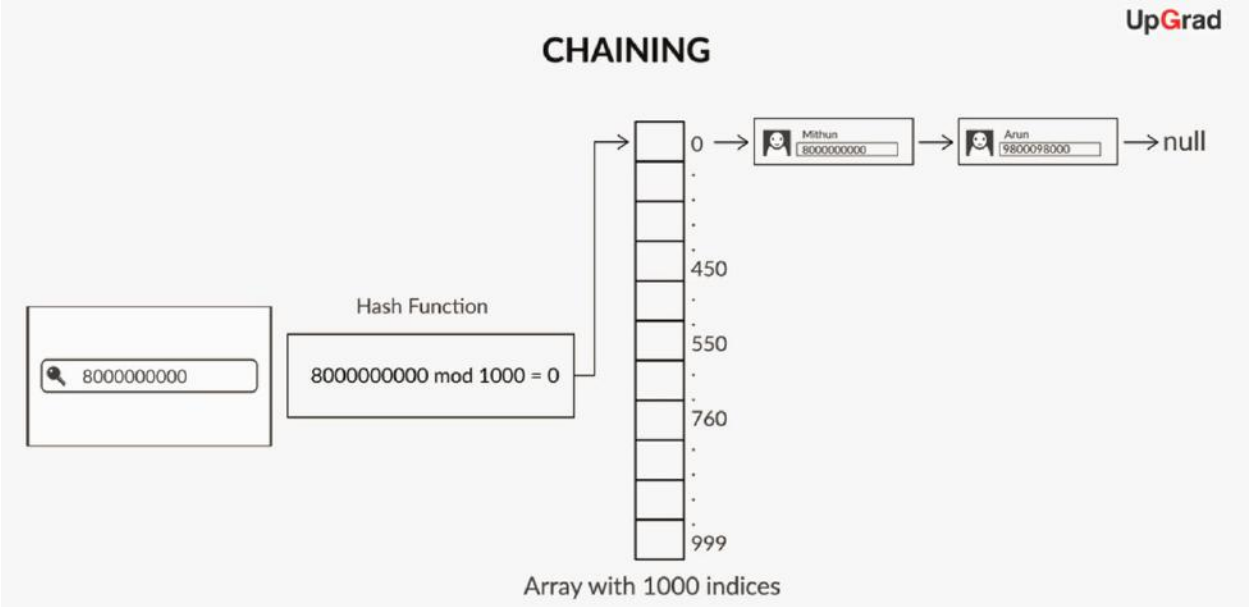


But again this method may fail under certain circumstances and still this function may not be collision proof.

Earlier the hash table was simply an array and each element in the array held one record. Now, in order to design a collision-proof hash function you should change the structure of this hash

table and that leads to designing the hash table as an array of linked lists.

Each location in the hash table will have a linked list of records. So you can go to any particular location and add one more element into the linked list. Therefore, if multiple keys hash on to the same location, you will end up storing both the records that hash on to the same location in two different nodes in the linked list.



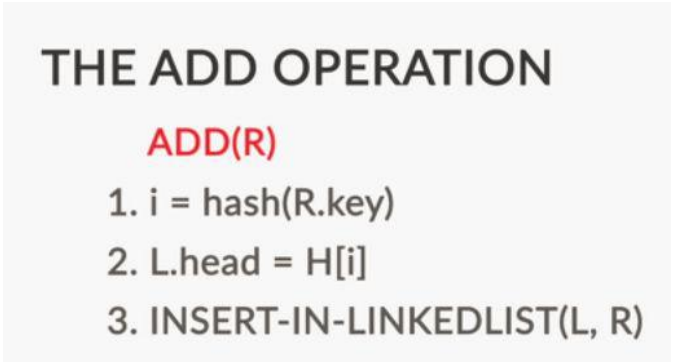
The memory requirement of a hash table, also referred to as its space complexity is O(n) where n is the number of records to be stored. The fundamental operations ADD, FIND and DELETE run in O(1) time for a hash table, as in the direct addressing method. This makes the hash table a preferred data structure for the implementation of a phone book. Collision is a drawback associated with hashing and this problem is alleviated using various methods like chaining and open addressing.

Performance Analysis of Hash tables with Chaining

You now have a hash table which is an array of n locations, each location is a linked list and if two values collide, i.e. if two keys are hashed onto the same location, then you will just add the records one after the other into the linked list.

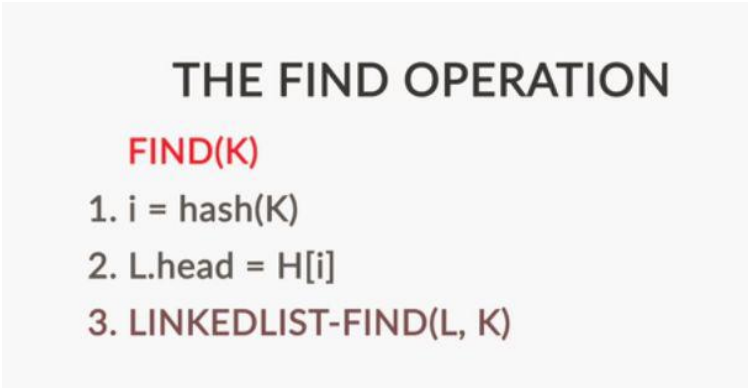
You have already seen how these functions, which are written with linked lists are slightly different because once you find the hash value, you only find the location and the record may lie in any node of the linked list of that corresponding location.

You can add a record in the linked list using the modified ADD operation as shown below:



Here each new record is allocated a specific location using the hash functions and its key. At that location, the record is stored in the first node of the linked list. The H[i] which was earlier just a simple value will now correspond to the head of the linked list. You will have to insert the new record at the first node of the linked list using the Insert-in-LinkedList operation. As you will be always adding the record to the first node, the add operation time will always remain constant, i.e. O(1).

Similarly, a FIND operation using linked list is shown in the figure below:



Earlier the hash function returned a location that was an index. So earlier you were simply returning H[i], where there was only one record in that location. Now H[i] is the head of the linked list which will correspond to L.head and therefore you will have to traverse through the linked list and then return a record using the LinkedList-Find operation.

As for finding the record you will be traversing through the linked list which could take a time of O(n) in the worst case.

Although the problem of Collision in hash tables is alleviated by using linked lists at each index location, in this process, the performance of the hash table is affected. Indeed, the running time of the FIND operation in a hash table with chaining grows to O(n) since you cannot return the array index directly. Rather, you need to search the linked list sequentially for the record.

Expected number of elements in linked list (n) = $\frac{Total\ number\ of\ items\ in\ the\ hash\ table\ (M)}{Size\ of\ the\ hash\ table\ (N)}$ **= (M)/(N)**

THE FIND OPERATION

FIND(K)

1. i = hash(K)
2. L.head = H[i]
3. LINKEDLIST-FIND(L, K)
   a. O(n) time in the worst case
   b. n = $\frac{M}{N}$
   c. Typically, we write O(1 + $\frac{M}{N}$)
   d. If M is close to N
   e. Then, $\frac{M}{N}$ is close to 1
   f. O (1 + $\frac{M}{N}$) is close to O(1)

The expected running time of the FIND operation in a hash table with chaining becomes O(1+M/N) where M is the number of stored records and N is it size, i.e. the number of indices in the hash table. O(1) represents the constant time required for hashing the key and O(M/N) represents the time taken to search the linked list sequentially. If M >>N, then the performance is adversely affected.

Sizing and Rehashing

Now that we have discussed about chaining in Hash tables, we move forward and review the performance factors of Hash Tables and discuss what is usually done to maintain a constant running time for the fundamental operations: Add, find and delete.

UpGrad

FIND OPERATION

FACTORS THAT DECIDE THE COST OF THE FIND OPERATION

1. Load factor, which is $\frac{M}{N}$
   - M = Number of elements inserted
   - N = Size of the table
2. Worst-case number in a linked list

WORST-CASE SCENARIO

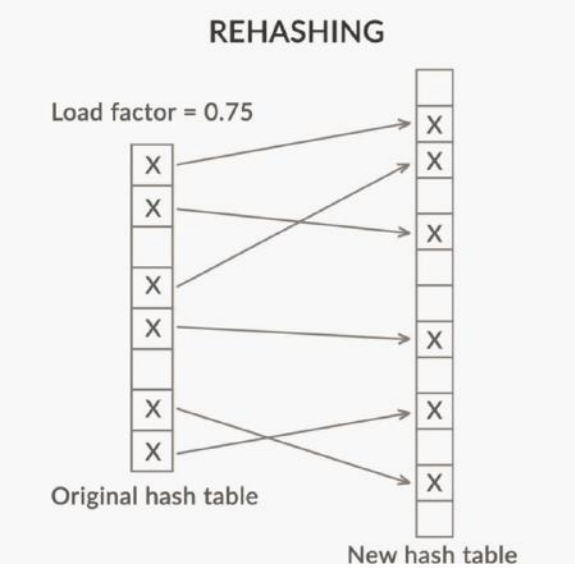
1. Cost of O $(1 + \frac{M}{N})$ for the FIND operation
2. Under the assumption that all the M elements are equally distributed
3. A good hash function ensures this
4. Some hash functions may distribute values unevenly
5. The values that arrive may be skewed
6. Some linked lists become long and some become short
7. The expected value of the length of a linked list is close to the total number of elements inserted

So the cost of the find operation depends on two things:

1. One is the load factor which is M over N where M is the number of elements inserted and N is the size of the table. This is one factor that determines the cost of the find operation.

2. The other factor is that the expected number M/N doesn't become the worst case number.

The running time of the FIND operation in a hash table is directly proportional to its load factor and hence, needs to be kept low. In practice, 0.75 is the load factor at which the hash table is sized and rehashed.



A new hash table double the original size is created and all existing records are rehashed into it. This helps in maintaining the constant running time of the FIND operation. To reduce the load on the Hash table, increase the size of the hash table. Therefore, M remains the same, N has been increased and hence, the Load factor $\frac{M}{N}$ correspondingly decreases.

You should not trigger rehashing quite often since every rehashing operation has a significant cost that is proportional to the current number of elements in the hash table.

Java API for Hash table

We have discussed a whole lot of theory regarding the dictionary ADT and its implementation using various data structures like sorted and unsorted arrays, linked lists and most importantly hash tables. Let's now look at how a real application like a contact list may be implemented in a programming language like Java.

UpGrad

A CLASS IMPLEMENTING INTERFACE CONTACTS

```
public class ContactList implements Contacts{
          public MobileNum findNum(String callName)
{//Method should be defined here}

          public void addEntry(String callName,
          MobileNum mb)
{//Method should be defined here}
}
```

The interface essentially gives a blueprint of the methods findNum and addEntry. The findNum method returns the phone number of the person that you are looking for on the contact list. The Add operation takes a contact list, adds a new entry with the name and a number and it returns a modified contact list and the user's understanding is that this is the list of entries.

Now the provider concerns can dictate which implementation to choose. You can choose one particular implementation. However, there could be alternatives for implementation. For instance, in this example, instead of a sorted list you could have used a hash table for the reasons show in the figure below.

UpGrad

REASONS FOR USING A HASH TABLE

1. The FIND operation runs for O(log n) time in a sorted list
2. But, O(1) time in a hash table
3. The operations are fast
4. An occasional delayed operation
5. In a hash table, the ADD operation is straightforward
   a. Time taken is O(1) in a hash table
   b. Time taken is O(n) in a sorted array

Java provides two APIs for hash tables, namely Hashtable and HashMap. These APIs provide several methods, the most important being 'put' and 'get'. 'put' is used to add records and 'get' is used to retrieve them using their respective keys. Recollect that records are considered as (key, value) pairs in the Java environment. In the phonebook example of looking up contacts using names, each name is a key and phone number is its value.

UpGrad

A CLASS IMPLEMENTING INTERFACE CONTACTS

```
import java.util.Hashtable;

public class ContactTable implements Contacts {
          private Hashtable namesAndNumbers;

          public ContactList(int initSize) {//Define
          method here}

          public MobileNum findNum(String callName){
          //Define the method here}

public void addEntry(String callName, MobileNum mb) {
          //Define the method here}
```

You have seen an implementation of the contact list using Hashtable class in Java. You can also use a HashMap implementation for the same.

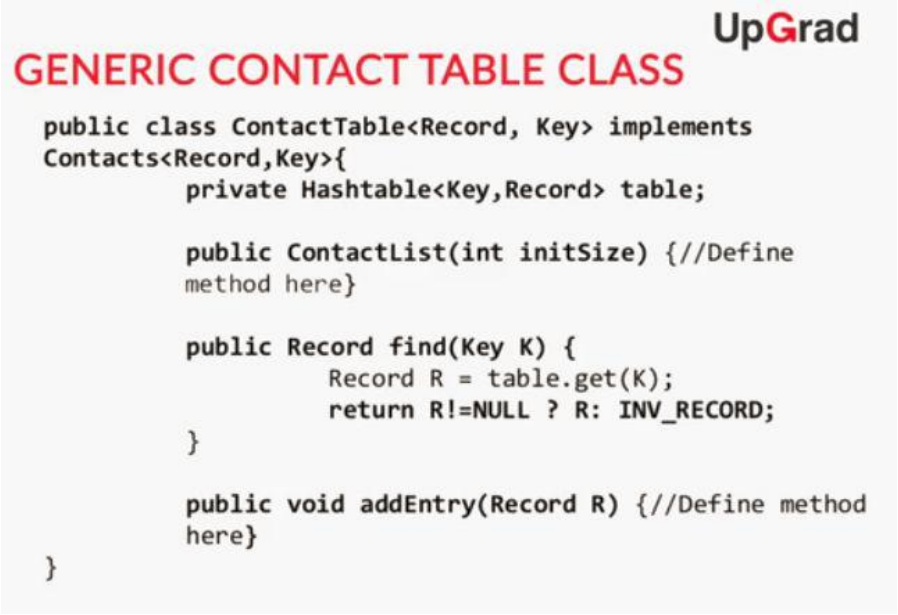
There are a few subtle differences between the Hashtable and HashMap APIs:

1. Hashtable is synchronized while HashMap is not.
2. Hashtable does not support null values while HashMap does.

ConcurrentHashMap is a Java class that provides thread-safe features like the Hashtable class while allowing concurrent get() calls. You may read further about the ConcurrentHashMap class and compare it with HashMap and Hashtable classes here.

Generic Hashtable in Java

Java supports generics which allows the developer to implement a dictionary to store and retrieve any type of key-value pairs using the Hashtable or HashMap APIs. In the last segment we looked at how to implement an ADT in Java and the example we used was that of a contact list. We then discussed how to generalize that ADT so that the contact list is not only in the context of a mobile phone but it could be any general contact list where the contact could be a list of email, phone, fax etc.



We have modified the contacts interface to accept record and key as type parameters. It means that when you implement a **public interface Contacts <Record , Key>** then the records could be any specific record **type** and that could be instantiated at the **point of use.** The interface above has two methods those methods operate on a generic type called record and a generic type called key.

Real-life applications require various types of keys and values like the Contacts app in your smartphone that supports both 'Search by Name' and 'Search by Number' features. These features may be implemented seamlessly in Java due to Generics in Java.

Let's recall some of the use cases for a generic interface in Java:

- Using generics, a hash table with keys of type Long(mobile numbers) and values of type String(names) is initialized as follows

  Hashtable numbersToNames<Long,String> = new Hashtable<Long, String>() ;

  The type parameters Long, String within the angle brackets <> declares the Hashtable to be constituted of keys of type Long and values of type String.

- For this hash table numbersToNames, a FIND function that looks up values of type String using keys of type Long is implemented as follows:

```
String FIND(Long callnumber){
        String R=numbersToNames.get(callnumber) ;
        return R!=null?  R : "Invalid Number";
      }
```

Note that it is not required to cast numbersToNames.get(callnumber) to String type because it is declared as String by the code generated by the compiler.

On the other hand, a hash table to implement the "Search by name" function would be as follows:

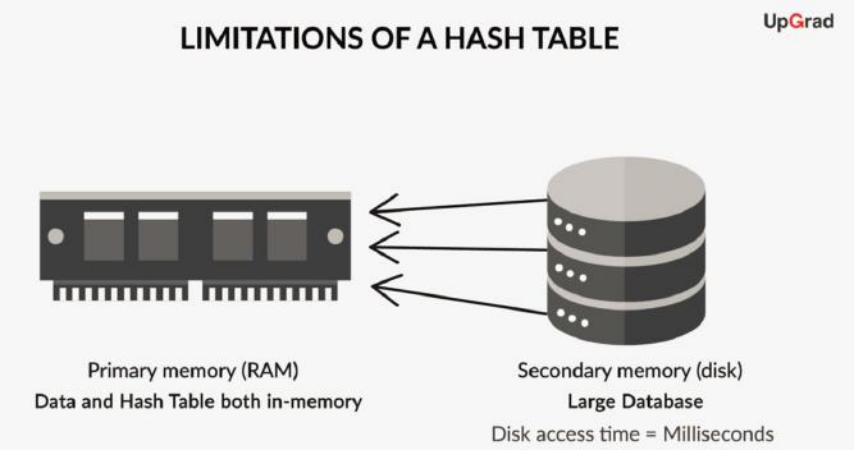
```
Hashtable namesToNumbers<String, Long> = new Hashtable<String, Long>();
Long FIND(String callname){
      R = namesToNumbers.get(callname);
      return R;
      }
```

Bloom Filters Introduction

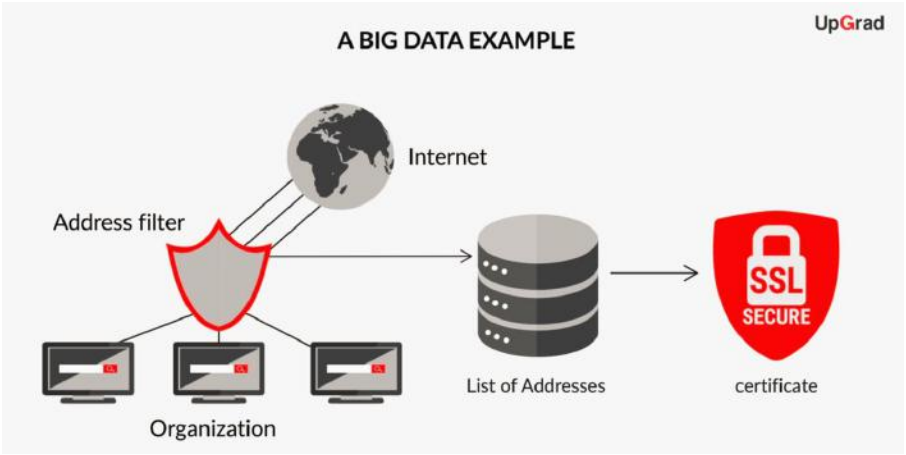
In this session, you were introduced to a compressed probabilistic data structure called Bloom filter. Bloom filters requires very less memory and are used to test the membership of records in a huge database.

Compressed Data Structures

So when you have situations where the **Hash table** gets really large then all the data cannot be stored in memory. Typically you store things in a hash table assuming that the data is all in **memory** and that the hash table is in memory. But when you have a large database, let's say of billions of records, you have to store the records on a disk.



Since you are unable to store them in memory, a portion of that is fetched at a time and then processed. This has a cost associated with it. The typical access time from hard disk is very large. In a disk, typically it's in milliseconds whereas in memory, the fetch time is in nanoseconds.



- So let's consider an example where you have a very large list of blocked addresses from which you don't want to receive any communication in your organisation.

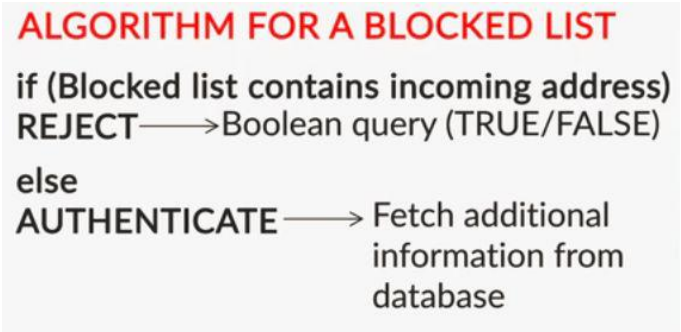
- So this list is fairly large because there are billions of addresses on the Internet and anybody could be sending packets which you don't want to receive in your network.

- So you have this large database on your network and the large database contains a list of addresses as well as authentication for valid addresses. For instance, you may have a certificate if the packet is coming from a valid address. The certificate allows you to check if the address is valid or you may have other information which says that information from this packet coming from a particular address is acceptable.

- So in your database, you have different categories of these addresses. Some of those addresses are blocked and if the address is blocked, you don't want to receive the packet from that address

To perform this task you can write a small algorithm, which is a 3 line algorithm which says **if an incoming address is in the block list then reject the packet. Else, authenticate the package using the certificate before passing it through.**

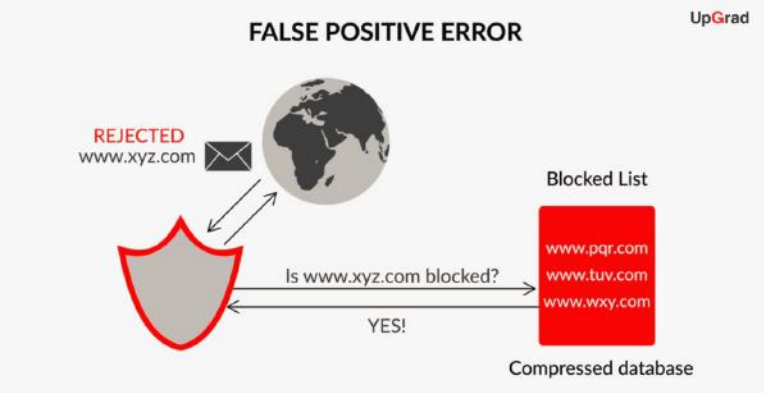
Now whenever an address is in block list, it hits the disk once since the address has to come from the hard disk, and when you say Authenticate, it hits the disk again. This two step operation can be converted to one by modifying the algorithm to “The first time you fetch the entry from the disk, store it in memory and then access it again and when you access it, take it from memory.” This operation is possible, so per packet you have to do one disk access. So, for n packets you will be requesting n disk accesses.



If you know that you only need to check whether the address is in the block list or not, this is a Boolean query and is going to return only a true or false value. So you don't need any other information from the database. But in case you need additional information from the database, such as owner information or a certificate, you will have to fetch this from the database and if the database is in the hard disc then you have no way of avoiding this second look up.

So the blocklist is usually a sublist of this total database, let's suppose you have only 10% of the address as your block list, now you don’t have to do a full query on the hard disk for every request for every package. Instead you can have a compressed database in memory. This

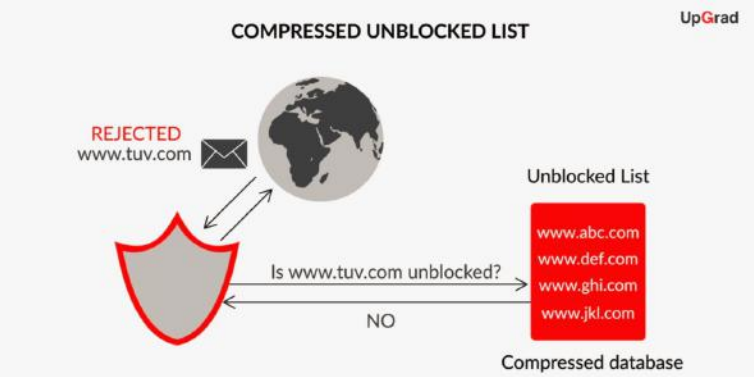
compressed database will only give you a yes or no answer. Now to do the boolean lookup, you will only use this database in memory and for that 10% of the requests, you will get a yes or no value. But you must keep in mind that if you try to compress the database you may lose information.



Assume that you have such a compressed database that answers the query with high probability accurately. It might be possible that sometimes it will give you a false answer saying that the address is in the block list although it is not in blocklist. So if there is a false positive, in that case you could reject packets which you are supposed to be allowing. To fix this consider the reverse scenario you can **modify** your earlier code as ***"****if address not in block-list then authenticate else reject."*

Now imagine a compressed database that has 90% of the items. Since you are compressing the database, you may lose some information. Due to the lost information, sometimes it is possible that you may get a false positive, i.e. it is going to say "the address is not in blocklist" even though it is a blacklisted operation. But if it says that the address is not on the blocklist then you are not immediately allowing the packet but first going to authenticate.

So when there is a false positive, you take a hit on your disk access time, because for a small percent of time you will be accessing the disk. But for a significant portion of the time, when it is not giving false positives, you can ignore the disk access and could do a reject.



Such a setup is a cost saving mechanism. When there is a scenario where you can actually construct the compressed table that may give some false positives, but most of the time it will give you the correct answer, and it does so in memory without having to store the entire database.

This solution is referred as a **Bloom filter** because it is typically used for filtering out accesses to the disk. When you have a large database in the disk and you can't store everything, you store a compressed database in the memory since your memory access is faster. So a Bloom filter is a solution for such a scenario where you have a large amount of data and quite often you have to go and hit the hard disk.

When you use a compressed database i.e. a Bloom filter and you have the compressed Bloom filter in local memory, you check and only go to the network when it is necessary. So at times when you are not going to the hard disk, you are not going to the network to access a distributed database and thus saving time. Hence, a reduction in running time is achievable using a Bloom filter which is a probabilistic database, which means it will give false positives occasionally but most of the time it will give a correct result and it will answer only yes or no queries rather than emitting the entire record.

Bloom filters are used to alleviate overheads due to disk and network fetch requests in data-intensive applications like web search. An important point to be kept in mind is that a Bloom filter is an array of bits, and hence a Bloom filter of size 1 MB is an array of 8×1024×1024 bits!

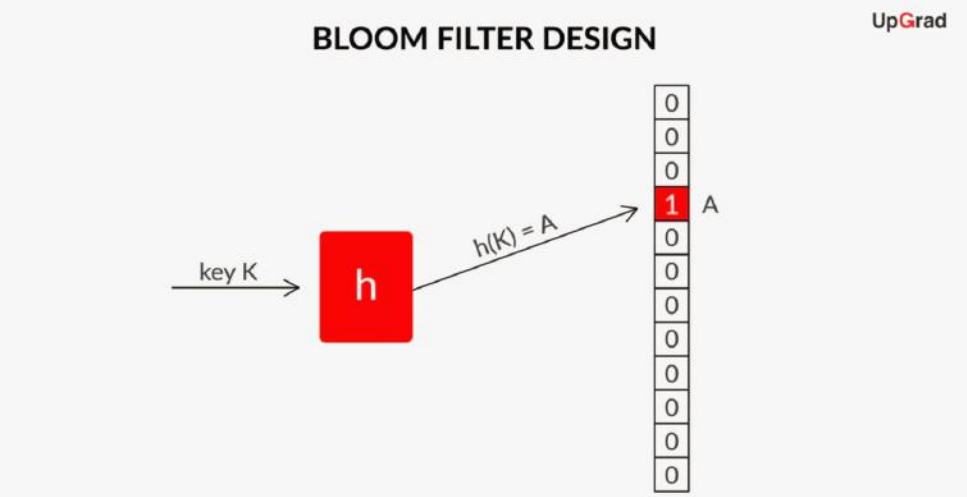
Bloom FIlter Design

By now you know that a Bloom filter is a probabilistic compressed data structure which only answers yes or no queries as opposed to Hash tables where the find operation can retrieve the whole record.

A Bloom filter is an array of bits coupled with multiple hash functions. Initially, all the bits are set to 0. When a record is added, its key is hashed by multiple hash functions and the bits

corresponding to their hash values are set to 1. If a record is to be looked up, its key is hashed by all the hash functions and the corresponding bits are checked if they are set to 1. If all the bits are set to 1, the Bloom filter returns YES or TRUE. If any of these bits are found to be 0, the Bloom filter returns NO or FALSE.

Essentially we use the Bloom filter to filter out queries which need not be considered further. As long as there are only false positives but no false negatives we can filter out queries and only when needed we go and check a secondary database.

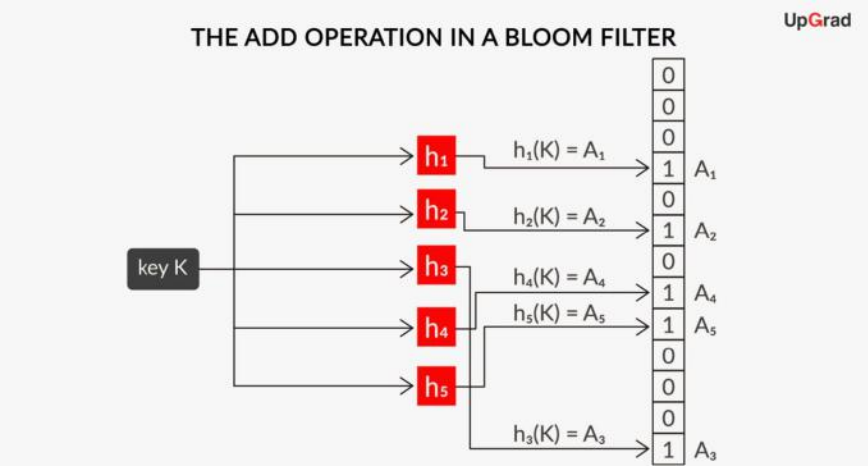


Now to ensure that one particular location does not get a 1 and many other values collide on to this and they all the end up turning to 1, we are going to use multiple Hash Functions.

Add Operation in Bloom Filter:

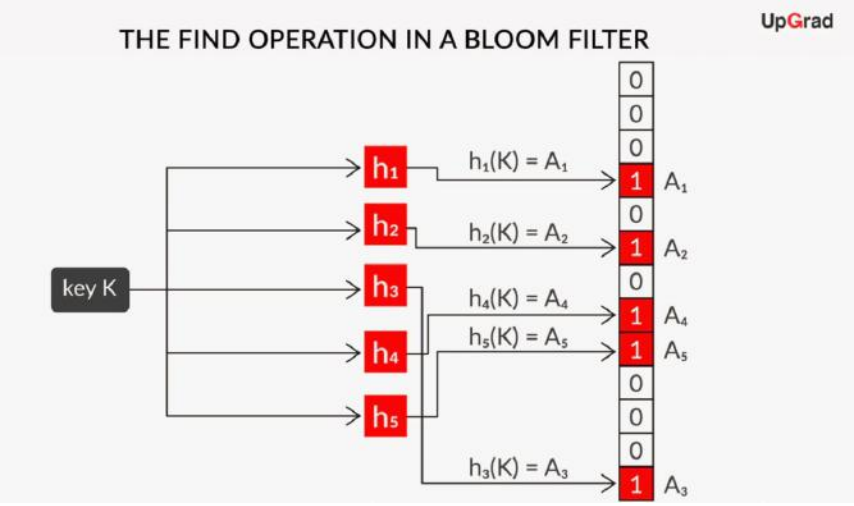
- Assume that you have 5 hash functions, H1, H2, H3, H4 and H5. When a key K is given to add a value, you compute the Hash Function 5 times using these different hash functions.

- Each of them is going to give a different location. H1 gives you A1, H2 gives A2 and then you get to A3, A4 and A5 accordingly. At all these locations you are going to set the bit to 1.

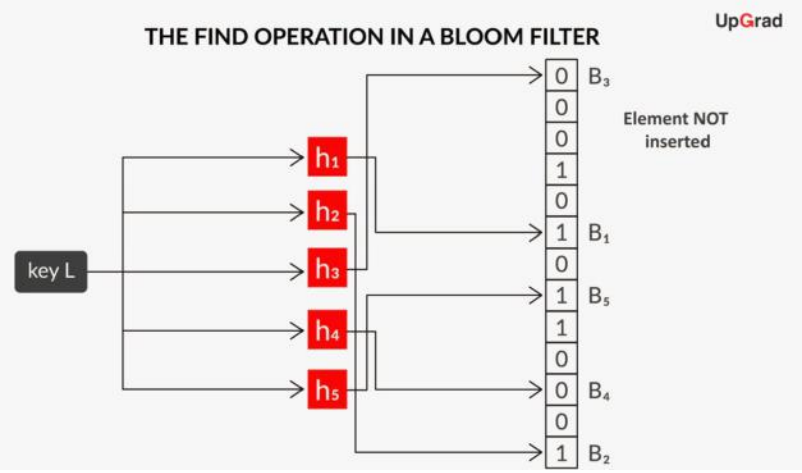


- All the 5 locations were obtained out of the same key using 5 different hash functions. All the hash functions are going to give you locations in the range of 0 to N -1 where you have N items in the Hash table and all these addresses or all these locations will be set to 1. Now this is the add operation.

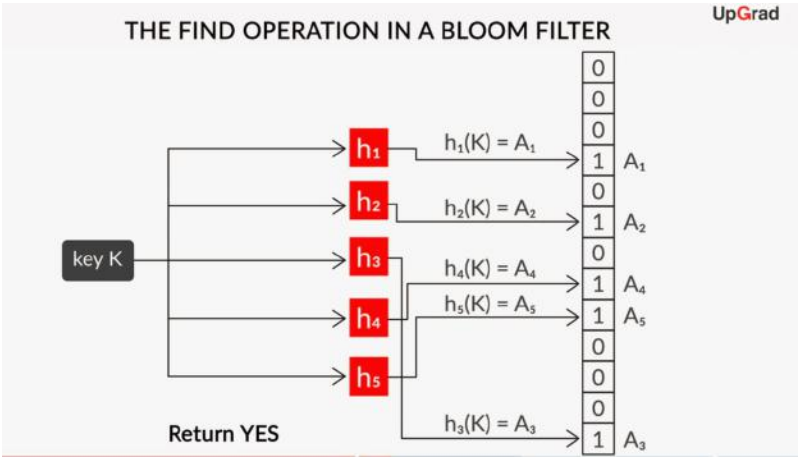
Find Operation in Bloom Filter:



Again compute the 5 hash functions for the same key as in the add operation. Each one will give you an address and you got 5 addresses. If all 5 addresses are 1, it implies that this particular value has been inserted.

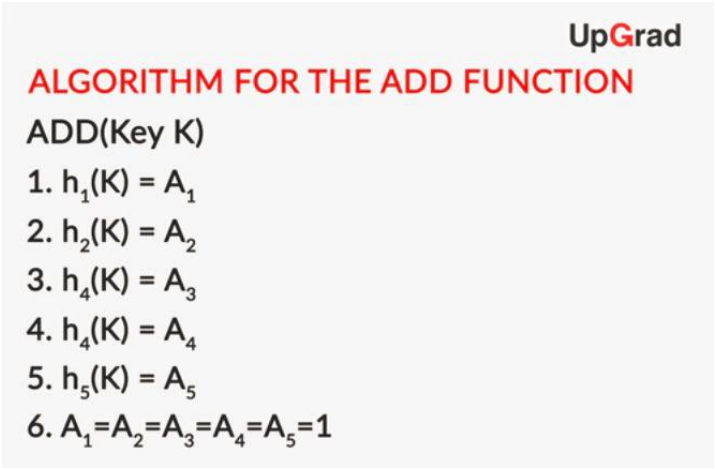


If one of the locations is zero, i.e. if you get a 0, it implies that this element was never inserted. So you compute 5 hash functions and if one of the locations is 0, then you know definitely that this value has not been inserted.



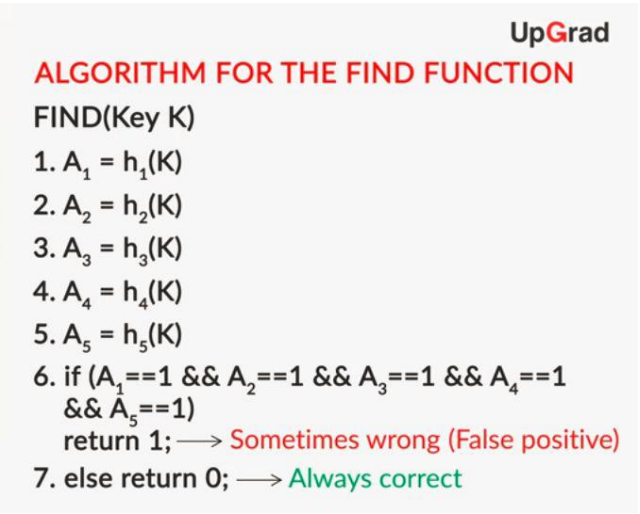
If all 5 values happen to be 1, it returns a YES answer. But it is not guaranteed that the value has been inserted. Because of collisions, some other key could have been inserted causing this to be set to 1, which might have generated a false positive. If you get at least one 0, then you know that this particular hash value has not been touched, which in turn means that this particular key has not been inserted. Hence, whenever you find a 0, you can confidently return a NO answer.

ALGORITHM FOR THE ADD FUNCTION



For an ADD(key k), you compute 5 hash functions. So you compute H1(K) = A1 and H2(K) = A2 and so on. Now you have these 5 addresses and want to set your table for these 5 different locations. For all the given 5 address you will set the value as 1. Hence, the time complexity of the add operations depends upon setting the bits corresponding to these hash values to 1. So the order for add functions is O(1).

ALGORITHM FOR THE FIND FUNCTION



The find operation is going to compute these locations and it's going to ask *"If all Ai's are 1 then return 1 else return zero."* (Here A[i] value is Hi(k)).
So you are going to compute all the hash values and ask *"If all the hash values are 1, then return 1, else return 0"*, returning 0 says that the element is not present. When you return 0, it is always correct. But when you are returned 1, you may be incorrect. This is referred to as a false positive and so this data structure may return a false positive. However, notice that the running

time complexity of the find operation is also O(1).

To summarize, a Bloom filter uses multiple hash functions to set certain bits to 1. When the FIND operation is called to search for a record, the exact same bits are checked if they have been set to 1, and if yes, it returns TRUE. Finally, False positive errors occur due to collision associated with these hash functions.

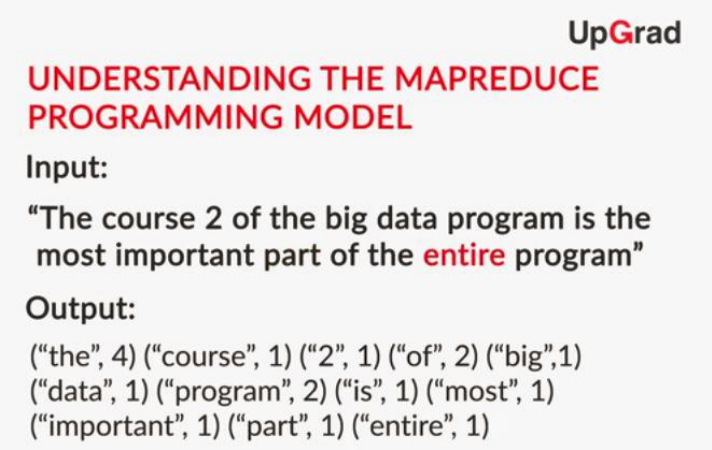
MapReduce Lecture Notes

In the previous segments you learnt about Apache Hadoop and its distributed file system, i.e. HDFS, and then proceeded to learn how to write programs that can be executed on a Hadoop cluster. Now in this module we discussed about MapReduce programming model, where you also learnt working principles of the MapReduce framework.

Understanding MapReduce Through the WordCount Program

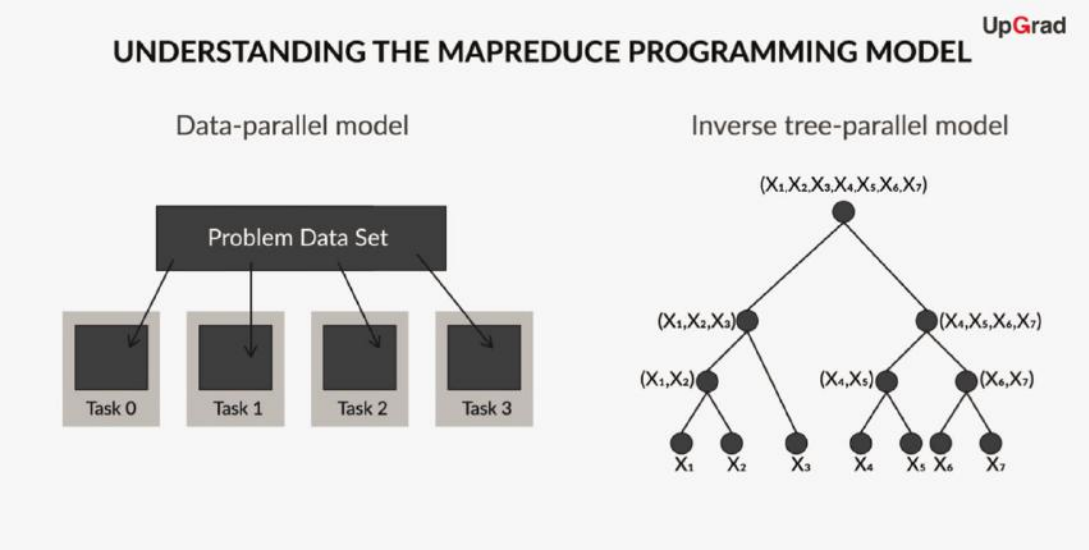
In this segment you understood the mapreduce programming model through the well-known example of counting the occurence of words in a document. The problem was as follows: You are given a huge list of documents and asked to list all the words along with their frequency.

For instance, the text “The course 2 of the big data program is the most important part of the entire program” should return (“the”, 4) (“course”, 1) (“2”, 1) (“of”, 2) (“big”,1) (“data”, 1) (“program”, 2) (“is”, 1) (“most”, 1) (“important”, 1) (“part”, 1) (“entire”, 1).



Since you are working in a distributed environment, different parts of the documents are stored across multiple nodes that are connected through a communication network.

The mapreduce programming model utilizes the data-parallel model during the map phase and the inverse tree-parallel model for aggregating results in the reduce phase.



Essentially, a mapreduce program executes as follows:

1. The master node initiates some number of map tasks depending on the number of file chunks.

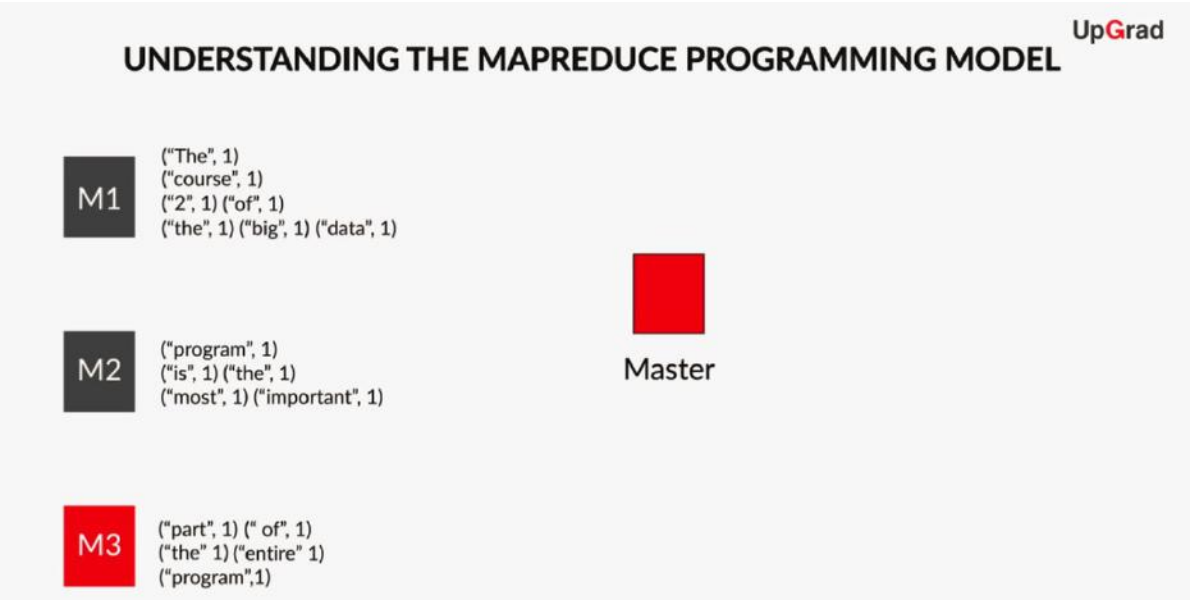
2. Each map task takes one or more file chunks as input and outputs a sequence of intermediate key-value pairs.

3. The master node collects all the intermediate key-value pairs from the map tasks and sorts them by key.

4. It then sends them to the reduce tasks in such a way that all key-value pairs with the same key go to the same reduce task.

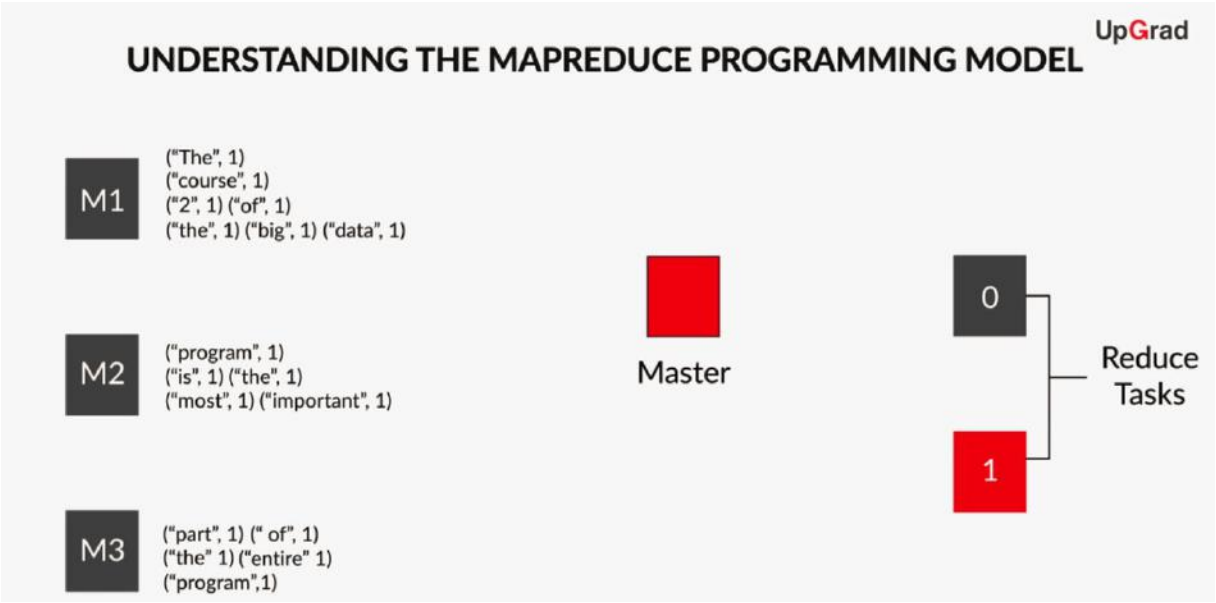
5. Each reduce task aggregates the entire set of values associated with a key, typically using a combination of binary operators.

Assume that you have three separate nodes storing each file chunk. In the first step, you write a map function that takes text as input and outputs key-value pairs of the form (word, 1) for each word in the chunk. The master node sees three file chunks and hence initiates three map tasks. These map tasks run in parallel on the three nodes. The first map task reads the input “The course 2 of the big data” and produces the following key-value pairs: (The, 1) (course, 1) (2, 1) (of, 1) (the, 1) (big, 1) (data, 1). The second map task reads the input “program is the most important” and outputs (program, 1) (is, 1) (the, 1) (most, 1) (important, 1) and similarly for the third map task.

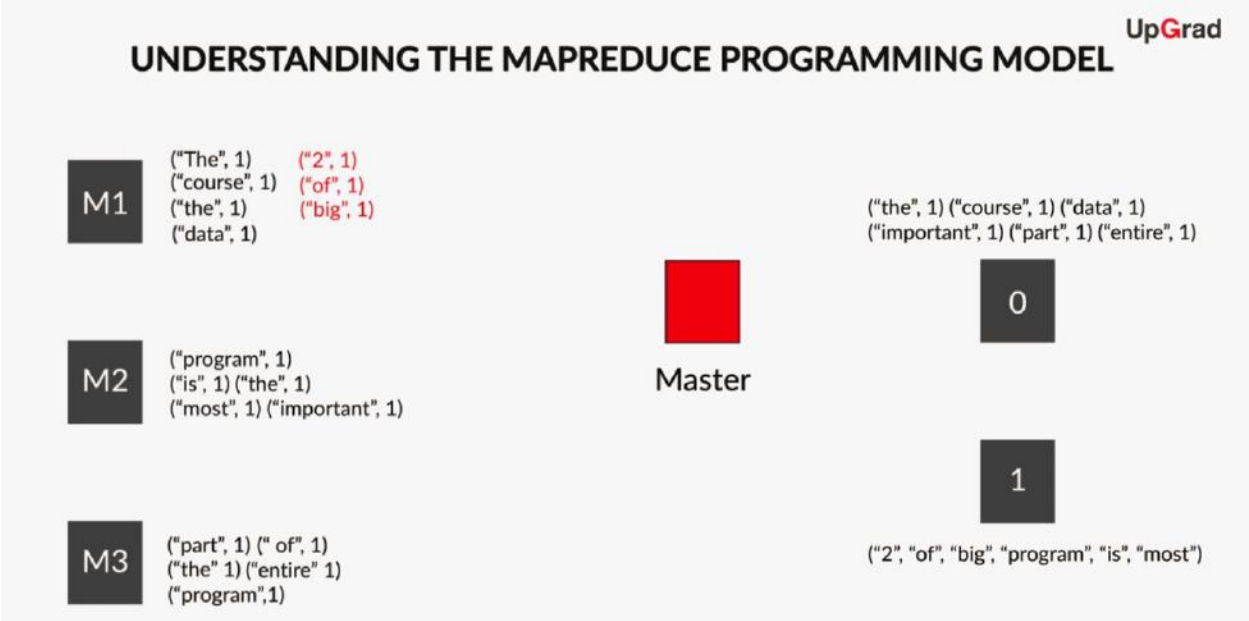


In the second step, the intermediate key-value pairs are grouped by the key. The master node allocates a certain number of reduce tasks as per the user's instruction, let's say it is r. It then uses a hash function to distribute the key-value pairs among the reduce tasks.

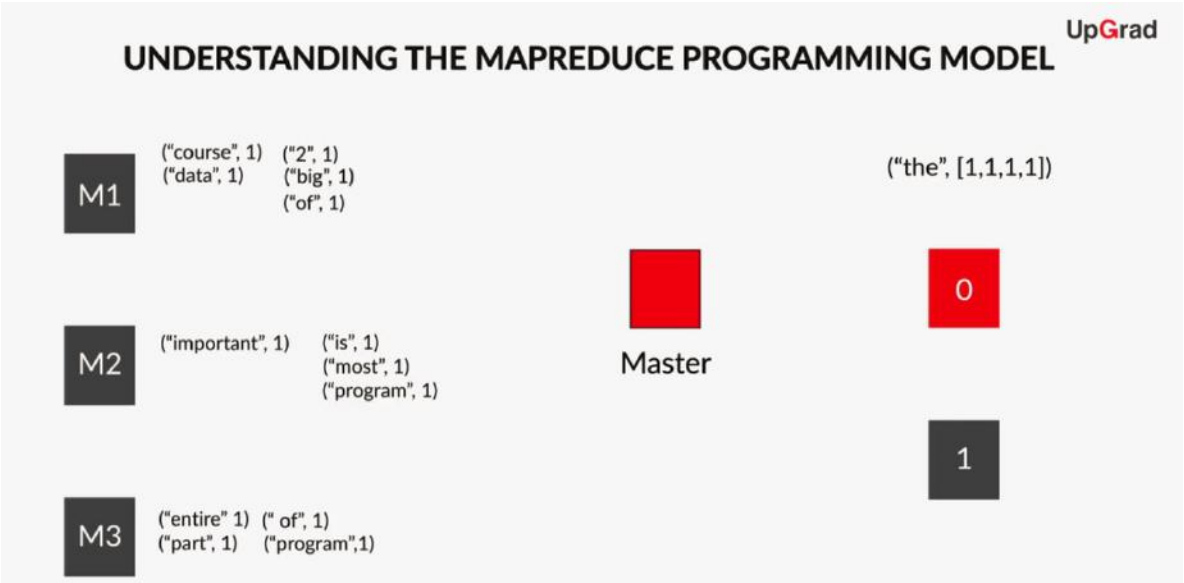
In our example, if there are 2 reducers numbered 0 and 1, the master node uses a hash code followed by a mod 2 operation to find the reducer corresponding to a key. The result of this operation would be similar to what is shown here: All the key-value pairs with keys "the", "course", "data", "important", "part", "entire" should be sent to reduce task 0 and the rest should be sent to reduce task 1.



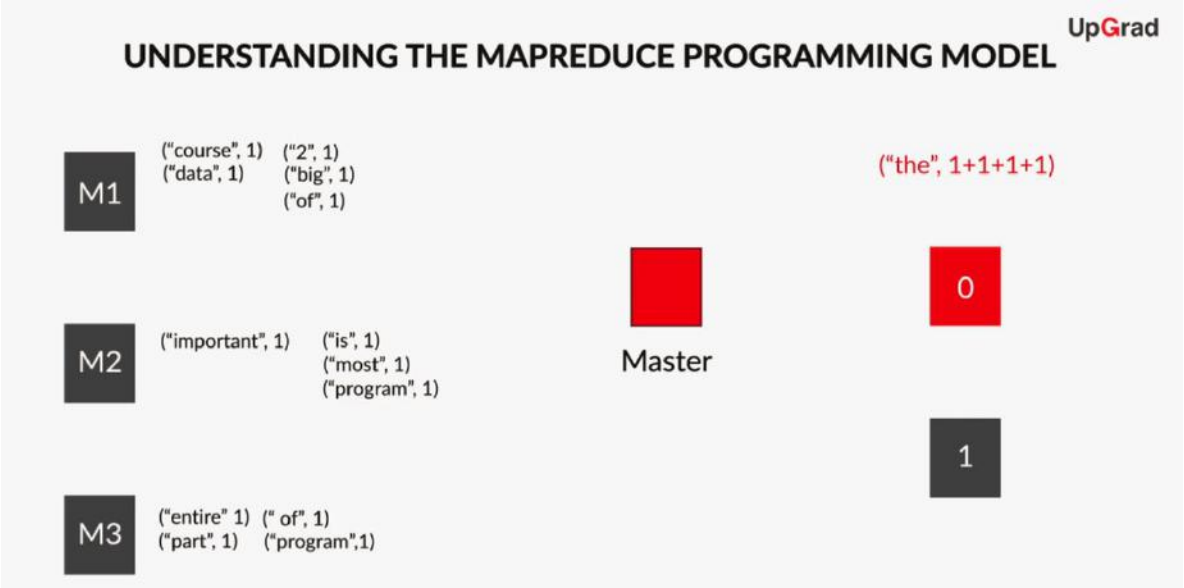
Based on this distribution, each node creates separate partitions to store the intermediate key-value pairs in its local disk. Node 1 creates two partitions. The first partition contains the key-value pairs, (“The, 1) (“course”,1) (“the,1”) (“data”,1) and the second partition contains (“2”,1) (“of”,1) and (“big”,1).



Similarly nodes 2 and 3 create two partitions each. The key-value pairs in each partition are then sorted according to their keys. As you can see, this naturally results in the grouping of the key-value pairs. Subsequently, the master node collects all the key-value pairs with a single key from all the map tasks, and sends the merged list to the respective reduce task.

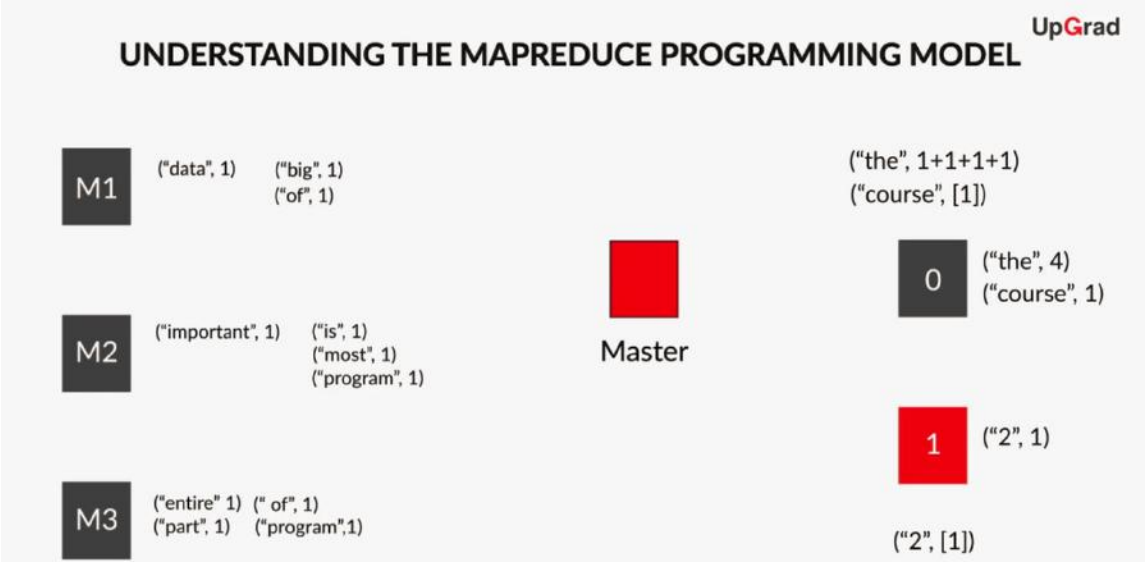


For example, the master node collects two key-value pairs (the,1) from the first map task, one (the,1) from the second map task and another (the,1) from the third map task, and merges them into a list of the form (the, [1,1,1,1]) and then sends it to its reduce task which is the reducer task, i.e. task 0 in our example.



there is a special name for the application of the reduce function to a single key and its associated values, which is called as a reducer. So, each reduce task executes multiple reducers.

For instance, reduce task 0 first executes a reducer that adds the four 1's associated with the key "the" and outputs ("the",4). For the next key "course", there is nothing to be added since it has only one value. Similarly, reduce task 1 executes a reducer that takes (2, [1]) as input and outputs (2,1) and so on.



In summary the following steps are involved in the execution of a MapReduce program:

1. The master node initiates a number of map tasks depending on the number of file chunks. All inputs and outputs in MapReduce are in the form of key-value pairs. As you will see later, the text document is also input in the form of key-value pairs. Each map task can take multiple file chunks in the form of key-value pairs as inputs and outputs a sequence of intermediate key-value pairs, which look like the following:

(K,V)→(K1,V1)(K2,V2)...
(K′ ,V′)→(K′ 1,V′ 1)(K′ 2,V′ 2)... .
.
(K′ ′ ,V′ ′)→(K′ ′ 1,V′ ′ 1)(K′ ′ 2,V′ ′ 2)...

Note that it is the user who describes the entire process of extracting useful information, i.e. these intermediate key-value pairs are extracted from the input data in the form of a Map function. Typically, there would be several common keys among these intermediate key-value pairs. In the next step, these intermediate key-value pairs that have common keys are grouped together.

2. The map tasks partition and sort all the intermediate key-value pairs by their keys. These are later merged by the Master node and sent to the reduce tasks in such a way that all key-value pairs with the same key arrive at the same reduce task in the form of a key and an iterable list of all its associated values.

Suppose that there are only three distinct keys, K1, K2, andK3, in the entire collection of intermediate key-value pairs that you obtained in step 1. Then, this step may be represented in the following manner:

(K1,V1)(K2,V2)... (K1,V11)(K1,V12)... (K1,[V11,V12,...])
(K′ 1,V′ 1)(K′ 2,V′ 2)...→ (K2,V21)(K2,V22)...→ (K2,[V21,V22,...])
(K′ ′ 1,V′ ′ 1)(K′ ′ 2,V′ ′ 2)... (K3,V31)(K3,V32)...
(K3,[V31,V32,...])

If you assume that there are two reducers, then the three groups shown above are distributed between these two reducers using a hash function on the keys. If hash(K1)=hash(K3)=0 and hash(K2)=1, then the groups with the keys K1 and K3 go to reduce task 0, and the group with the key K2 goes to reduce task 1.

3. Each reduce task aggregates the entire set of values associated with a key, typically, by using a combination of binary operators. Again, it's the user who describes this process in the form of a Reduce function. By denoting the binary operator with an asterisk (*), this step may be represented in the following manner:
   Reduce task 0 (K1,[V11,V12,...])→(K1,V11∗V12∗...)
   (K3,[V31,V32,...])→(K3,V31∗V32∗...)
   Reduce task 1 (K2,[V21,V22,...])→(K2,V21∗V22∗...)

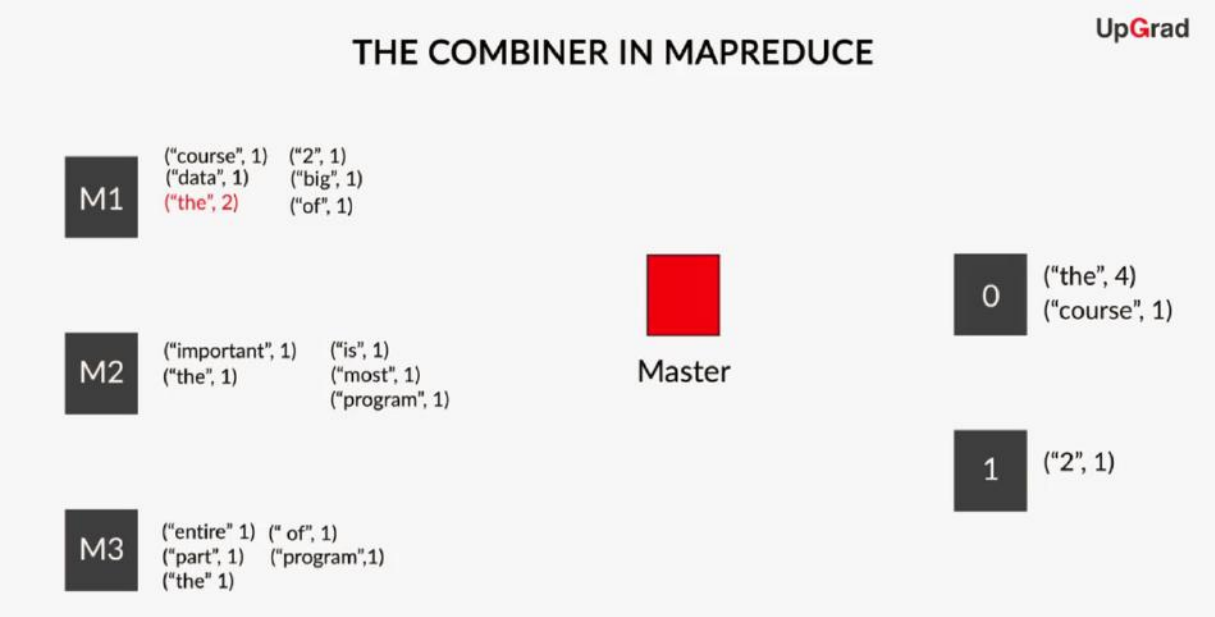
Finally, the output of all the reduce tasks is stored on HDFS.

The Combiner

In the previous segment, you saw that all the intermediate key value pairs are transferred to the reduce tasks through the master node. This doesn't look daunting for a small example that you have considered here. However, if you were to perform word count at the scale of millions of documents, these intermediate key-value pairs may themselves be huge in number.

The mapreduce framework provides us a way to achieve the same. Instead of transferring the entire set of intermediate key-value pairs, you can perform some part of the reduce function in the map task itself.

Consider the first map task that produced the intermediate key-value pairs ("The", 1) ("course", 1) ("2", 1) ("of", 1) ("the", 1) ("big", 1) ("data", 1).



Instead of passing two separate tuples ("the",1) you can add their values and then transfer a single tuple ("the", 2) to the reduce task. If your document had a thousand occurrences of the word "the", then you end up saving a lot of time by performing this intermediate aggregation. This step is typically referred to as the combine phase of MapReduce framework.

However, the combiner works only if the reduce function is commutative and associative. The Combiner dependencies are as follows:

- The reduce function must be independent of the sequence in which it's executed.
- If the reduce function consists of the averaging operator, then the combine phase fails to work. This is because, the averaging operator is not associative.

A combiner may be used even for the non-commutative and non-associative reduce functions by writing a separate combiner function.

So in summary the Combiner, also known as the semi-reducer, is an optional class in the MapReduce style of programming. It is particularly useful when the reduce operation is commutative as well as associative.

- We say that a reduce operation Reduce() is commutative if —

Reduce((K,[V1,V2]))= Reduce((K,[V2,V1])), for all possible key-value pairs.

- We also say that Reduce() is associative if —

Reduce((K,[V1,V2,V3]))= Reduce((Reduce((K,[V1,V2])), Reduce((K,[V3])))

= Reduce((Reduce((K,[V1])), Reduce((K,[V2,V3]))

In other words, we say that a reduce function is both commutative and associative if it produces the same result irrespective of the sequence in which it is executed.

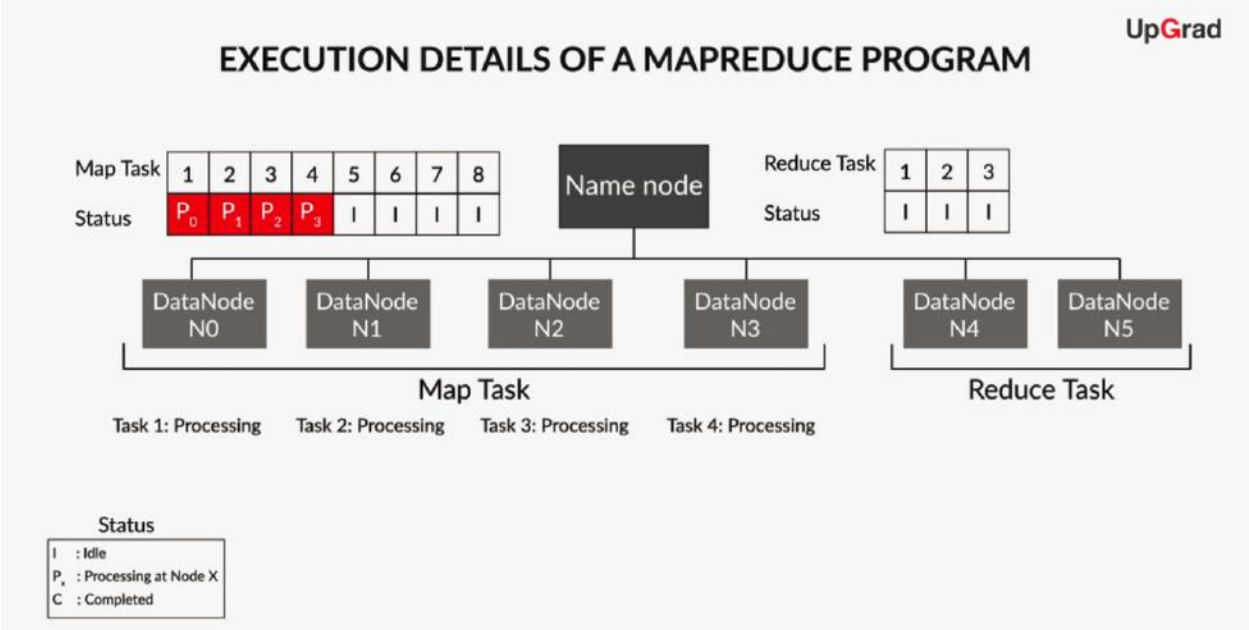
You saw that we could use a combiner for the WordCount program since the reduce function would add up the values associated with each key, addition being both commutative and associative.However, if the reduce function were to take an average of the values, then you need to write a separate combine function.

Job Scheduling and Fault Tolerance

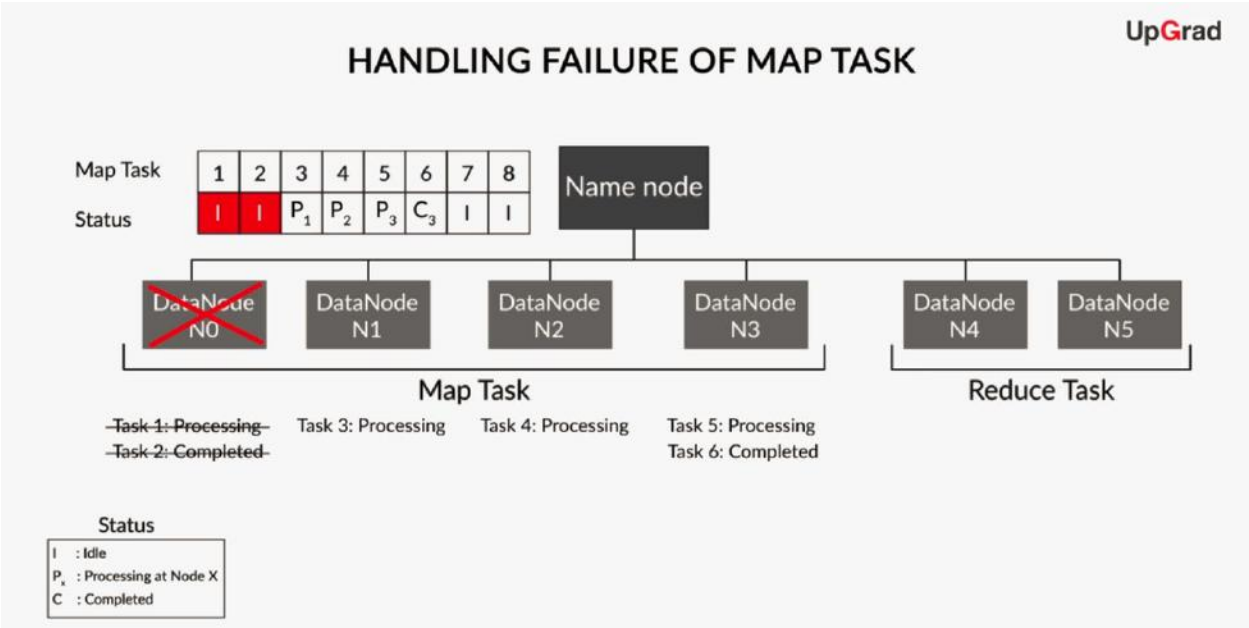
In this session you understood the execution details of a mapreduce program. In the previous module, you looked at the HDFS file system. In particular, you saw that there are multiple data nodes(slave) and a single namenode(master) that coordinates with and manages the data nodes.

Given a mapreduce program, the master node usually assigns either a map task or a reduce task to a worker node, but not both. The master node is responsible for creating a number of map tasks and reduce tasks. Each of these numbers is specified by the user program.

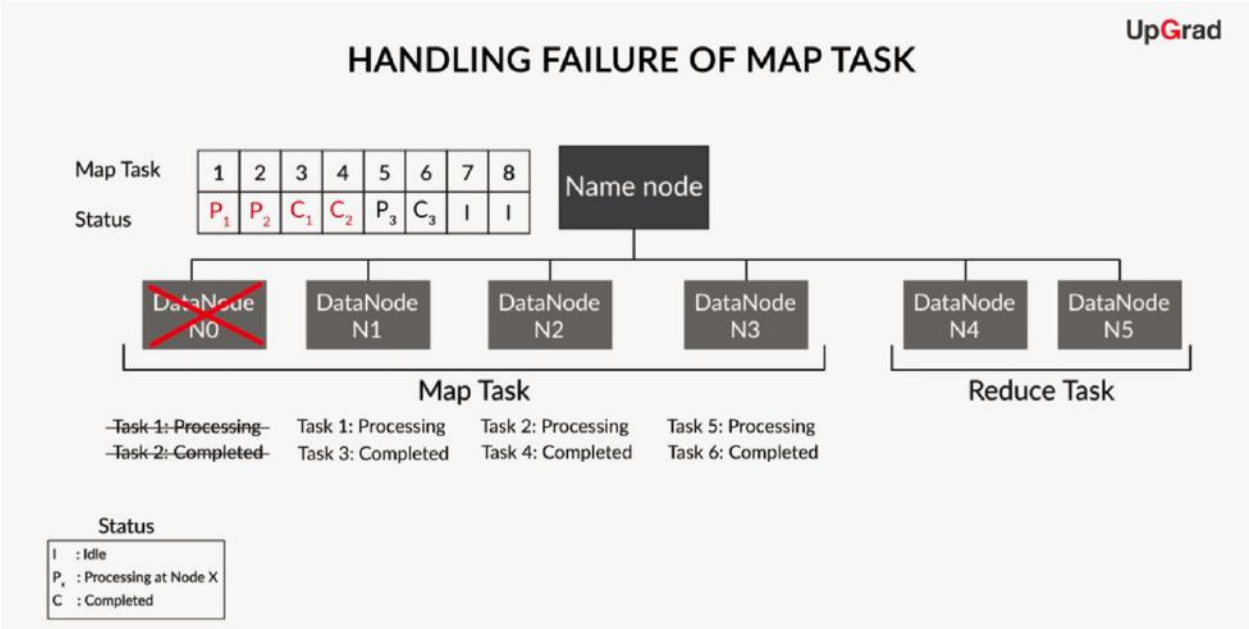
The master node maintains a queue of map and reduce tasks along with their status, namely, IDLE or PROCESSING AT NODE X or COMPLETED. Depending on the availability of worker nodes, it assigns each idle task to a node.



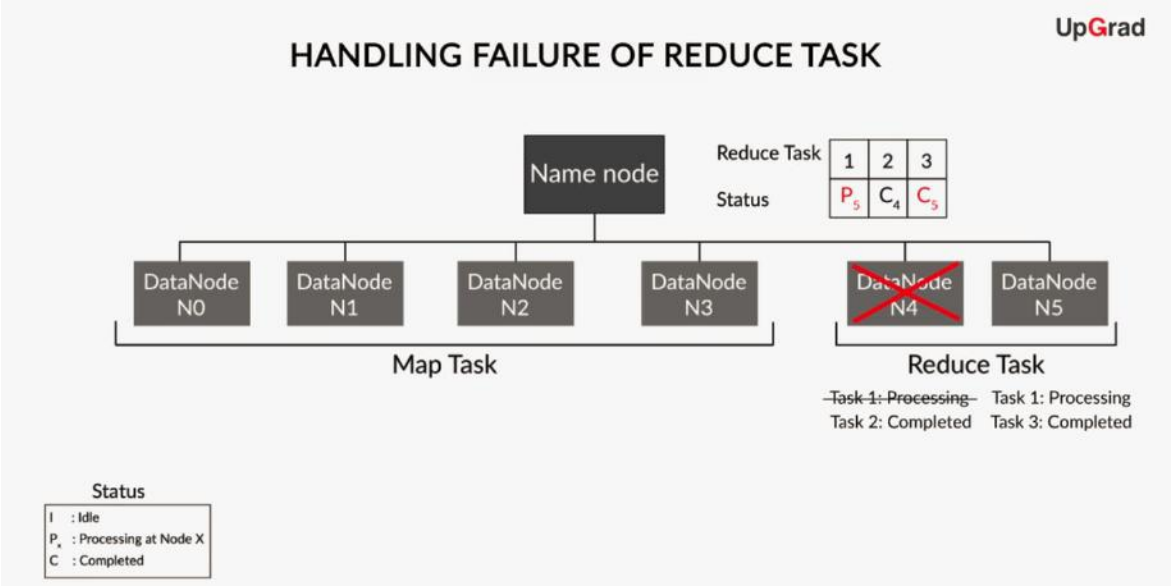
Now suppose a worker executing a map task fails. The master not just resets all the map tasks being processed by the same worker to idle, it also resets the tasks that were completed by that particular node. This needs to be done since the intermediate key-value pairs are stored in the local disk of the worker node and hence, they are no longer available after its failure.



Therefore, all those map tasks are added to the master's queue and are then assigned to other nodes when they become available. Further, the master also informs the respective reduce task about the updated node from where it has to receive its input.



In case a node executing a reduce task fails, it is much simpler for the master to handle. It only has to reset the tasks that are being processed to IDLE since the results of the completed reduce tasks are stored in the HDFS file system and hence, they are still available. These tasks are later allotted to one of the healthy nodes when they become available. Typically, the number of map tasks is kept higher than the number of worker nodes.



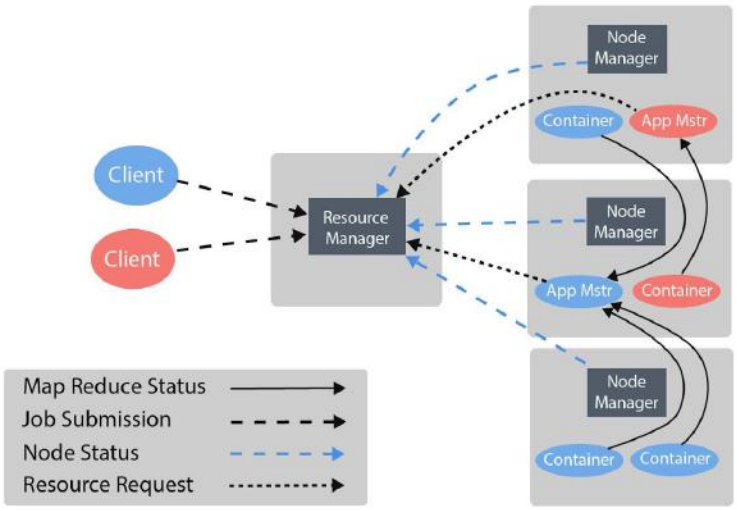
To be more precise, the number of map tasks is kept close to the number of distinct file chunks. This helps in better processor utilisation, especially in case of node failures.

If the number of map tasks was equal to the number of worker nodes, then a failure would require the master node to wait till one of the healthy nodes becomes available to assign the failed task. This would result in unbalanced and non-optimum processor utilisation.

So in summary you learnt about the execution process in the context of MapReduce 2 in Hadoop 2.0, also known as Yet Another Resource Negotiator (YARN). Please note that the basic idea remains unchanged. However, there are a few changes with regard to the nodes performing the scheduling operations. The details of MapReduce 1 in Hadoop 1.0 may be looked up here.

YARN also follows the Master-Slave architecture. The master is called ResourceManager, and the slave is called NodeManager. Each cluster has exactly one ResourceManager and several instances of the NodeManager. All instances of NodeManager are usually run on the same servers running the HDFS DataNodes. However, this is not a mandatory requirement. Similar to HDFS, the NodeManagers send heartbeat signals to the ResourceManager, thereby communicating that they are alive. A significant difference between MR 1 and MR 2 is that the ResourceManager performs only resource management and delegates any work related to

scheduling to an ApplicationMaster that runs on a slave machine. The ApplicationMaster too sends regular heartbeat signals to the ResourceManager.



The details of a MapReduce execution are listed below:

1) A client submits a MapReduce application to the ResourceManager. The ResourceManager has a component called ApplicationManager that checks the request for duplicates, looks up if there exist nodes with the required resources to even run the ApplicationMaster, and then forwards them to the Scheduler, which is another component of ResourceManager.

2) The ResourceManager then launches an ApplicationMaster on a slave node. From this point onwards, it is the ApplicationMaster's responsibility to schedule the required number of map and reduce tasks.

3) The ApplicationMaster registers itself with the ResourceManager and then requests for resources to perform the MapReduce job.

4) The ResourceManager, through the heartbeat signals from NodeManagers and the existing ApplicationMasters, obtains complete information about the available resources. The resources such as memory, network bandwidth, and CPU cores are abstracted in YARN in the form of containers. You can think of containers as workers performing the tasks (map and reduce) that were explained by the professor. So, the

ResourceManager informs the ApplicationMaster about the completed containers, newly allocated containers, and the current state of available resources.

5) The ApplicationMaster then sends signals to launch the application on the allocated containers. In other words, it launches a number of map and reduce tasks (according to the user inputs or by default) and sends them the required set of instructions contained in the user program.

6) The ApplicationMaster continuously monitors the state of the launched containers through the ResourceManager or the individual NodeManagers and acts in case of node failures.
    a) If a node with a container running a map task fails, the ApplicationMaster not only resets all the map tasks that are being processed by the same node to become idle, it also resets the tasks that were completed by that node. This is necessary because the intermediate key-value pairs are stored on the local file system of the node, and hence, they are no longer available after it encounters a failure. Therefore, all these map tasks are added to the ApplicationMaster's queue and are then assigned to other nodes when they become available. Furthermore, the ApplicationMaster also informs the respective reduce task about the updated node from where it has to receive its input.

    b) In case a node executing a reduce task fails, it is much simpler for the ApplicationMaster to handle. It only has to reset the tasks that were being processed by that node to become idle, since the results of the completed reduce tasks are stored in the HDFS file system, and hence, they are still available. These tasks are allotted to healthy nodes later when they become available.

    c) In case the node running the ApplicationMaster fails, the ResourceManager launches another instance of it on a healthy node. However, there is no functionality as yet to resume its state before failure. In other words, the MapReduce job needs to be started all over again.

    d) In case the master node hosting the ResourceManager fails, the entire cluster becomes unusable until it is restarted, i.e. the ResourceManager is the Single Point Of Failure (SPOF) in a Hadoop cluster.

7) On successful completion of the MapReduce job, the ApplicationMaster notifies the ResourceManager. The ResourceManager keeps a log of the execution including the location of the final output, which may be read by the client.

Typically, the number of map tasks is kept higher than the number of DataNodes. The default number of map tasks equals the number of distinct file chunks. The user can change the default setting for the input split size by modifying it. For instance, you may specify an input split size of 64MB, instead of the default block size of 128MB. And the number of map tasks spawned would then become double the default number. This helps in better processor utilisation, especially in case of node failures.

If the number of map tasks was equal to the number of nodes, then a failure would require the ApplicationMaster to wait until one of the healthy nodes becomes available to get assigned to the failed task. This would result in an unbalanced and non-optimum processor utilisation. On the other hand, if the number of map tasks was higher, then all the map tasks residing on a failed node can be distributed among the available nodes, thereby maintaining a higher level of resource utilisation.

However, you need to remember that the number of reduce tasks is always kept low. The default is to have one reduce task. This is because the map tasks create intermediate files or partitions on their local file system corresponding to the number of reduce tasks. A large number of reduce tasks will result in a large number of such partitions, which is undesirable in HDFS.

MapReduce programming skills may be developed only through practice. You may watch a few code demonstrations and download the java files here and here.

Lecture Notes

PIG

This module deals with the usage of PIG, its use cases, features and its advantages over mapreduce. The first session deals with the introductory part, where you were introduced to PIG and its basic data types. The second session deals with the demonstration of some commonly used PIG commands through a PIG script. The third and the last session includes an industry use case demonstration where some of the problem statements, related to the given dataset, were solved.

Why Pig?

Some of the limitations of mapreduce are:

- Expert-level knowledge of JAVA
- Expertise in optimisation of mapreduce jobs
- A lot of effort in terms of time and lines of code to create a simple job using Java.

PIG successfully overcomes these limitations.

Features of PIG

The major features of PIG are as follows:

- In PIG instructions are written in a language called Pig Latin.
- With a little bit of practice, Pig Latin can easily be understood by professionals with little or no programming experience.
- PIG internally converts all the converted into optimised mapreduce jobs. So, you don't have to worry about the optimisation of the jobs.

- PIG saves a lot of effort in terms of time, which professionals used to invest in coding complex MapReduce tasks.
- The execution time of PIG scripts is higher than mapreduce jobs considering the fact that PIG internally uses mapreduce to compute results.

Data types in PIG

The data types in PIG can be divided into two major categories:

1. Scalar data types
2. Complex data types

Scalar Data types:

These consist of the primitive data types which are similar to primitive data types in any other programming language. Some of the scalar data types used in PIG are:

- *chararray*: It is the same as the String data type in other languages. It is used for representing names, addresses, and so on.
- *float*: It is used for representing real numbers for variables like weight, marks obtained, etc. The float data type can store 32-bit floating-point numbers.
- *double*: It is also used for representing real numbers but for a range greater than that represented by the float data type. For example, Total Sales, Total Expenses, etc. The double data type can store 64-bit floating-point numbers.
- *int*: Used for representing integral values such as age and count of students.
- *bytearray*: It is the default data type of pig. If no data type is declared, then Pig, by default, assigns bytearray as the data type.

Complex Data types:

Complex data types are the collection of primitive data types. This consists of the following:

- *Map*: It is a set of key-value pairs. The keys and values are separated by '#' symbols and are enclosed in square brackets. The key-value pairs are separated by comma (,).
  For example, [name#david, age#20, place#chicago]
- *Tuple*: It is an ordered collection of items. It is enclosed within round brackets.
  For example, (david, 20, chicago)
- *Bag*: It is a collection of tuples or bags. The tuples are enclosed in curly brackets.
  For example, {
  (david, 20, chicago),
  (lettice, 28, london)
  }

Execution modes in PIG

There are two execution modes in PIG:

1. **Local Mode**:
   The local mode execution in PIG requires access to just one system. It does not require any hadoop services running in the background. Also, it uses the local file system for loading and storing the data. Execution in local mode can be done using the command **pig -x local**.

2. **Mapreduce mode**:
   The mapreduce mode execution requires access to a distributed hadoop cluster or a virtual machine where hadoop is running in pseudo-distributed mode. It requires all the hadoop services running in the background. Also, it uses HDFS for loading and storing the data. Execution in mapreduce mode can be done using the command **pig**.

Grunt shell

PIG code can be run in two ways:

- You can execute a PIG script file. The total PIG code is written in this file. And the commands written in this file are executed in a sequential manner.

    - In local mode, a script file named **pigscript.pig** can be run using the command **pig -x local pigscript.pig**
    - In mapreduce mode, a script file named **pigscript.pig** can be run using the command **pig pigscript.pig**

- You can run PIG commands one at a time using interactive mode. The shell that executes pig commands is known as grunt shell.

    - In local mode, the grunt shell can be invoked using the command **pig -x local**.
    - In mapreduce mode, the grunt shell can be invoked using the command **pig**.

Basic commands

1. **Load**:
   Used to load a data set into a relation.
   E.g. **a= load 'users/data/data.txt';**
   This command will load all the data that is written in the file named **data.txt** at the location **users/data** into the relation named a. Relation in PIG is just another name for what we call as variables in other programming language. They store the data.

2. **Load using PigStorage**:

   By default, PIG uses **tab** ('\t') as the default delimiter for the columns. PIG will look for the delimiter character in each line(row) of the file and split the line on the basis of the delimiter. In case we want to use some other character to separate out the columns in PIG, we can specify it by using **PigStorage.**

   Eg. **a= load 'users/data/data.txt';**

   This command will use **tab** ('\t') as the delimiter while,

   b= load 'users/data/data.txt' using PigStorage(',');

   This command will use **comma** (',') as the delimiter.

3. **Loading with schema**:

   - You can specify schema in PIG using **AS** keyword followed by the schema.

     - Eg. **a= load 'users/data/data.txt' using PigStorage(',') as (ID:int, Name:chararray, age:int, salary:double);**
     - In the above example, we have specified four columns namely, ID, Name, age and salary. These columns have data types as integer, chararray, integer and double respectively.

   - In case, you only define the name of the columns in the load command and not the data types, PIG will assign bytearray as the data type for each column

     - Eg. **a= load 'users/data/data.txt' using PigStorage(',') as (ID, Name, age, salary);**
     - In the above example, we have specified four columns namely, ID, Name, age and salary. All the four columns have data type as bytearray.

4. **Describe:**

   Describe command can be used to get the schema of any relation.

   Eg. **a= load 'users/data/data.txt' using PigStorage(',') as (ID:int, Name:chararray, age:int, salary:double);**

   DESCRIBE a;

   Here, **DESCRIBE a;** will print the schema of relation a on the screen which is **{(ID:int, Name:chararray, age:int, salary:double)}**

5. **Store:**

   Store command is used to store the data at a particular location. The command takes the path to a folder as an input. The data is stored inside that folder as a **part-r** file.In case, the specified folder location already exists, the script will throw an error and will not store the data.

   Eg. **STORE a INTO 'data/output/';**

   This command will store the data contained in relation a inside the data/output folder.

6. **DUMP:**

   DUMP command will print the contents of a relation on the screen.

   Eg. **DUMP a;** will print the contents of relation a on the screen.

7. **FILTER BY:**

   Filter by command will filter the rows of a relation on the basis of a predicate.

   Eg. **b= Filter a by name='Mohit' AND age>20**

   This command will select all the rows from relation a where the value in the column **name** is equal to **Mohit** and value in the column **age** is greater than **20**.

8. **FOREACH GENERATE:**

   Foreach generate command will loop over the relation and generate the columns or the specified transformations over a column.

   Eg. **b= FOREACH a GENERATE Name, 2018-age AS BirthYear**

This command will create a new relation with two columns namely **Name** and **BirthYear.** The values in the column **Name** contain the same values as in column **Name** of relation a while the values in column **BirthYear** are calculated as the difference between 2018 and the value in **age** column of relation **a.**

9. **ORDER BY:**

Order by is used to arrange the rows of a relation in a specific order on the basis of a column.

- The default order is ascending order. Also, if you want to explicitly specify the order as ascending order, you can use the keyword **ASC**.
    - Eg. **b= ORDER a by age ASC;**
    - This command will arrange the rows in relation a in ascending order on the basis of age. That is the youngest person's record will appear first.
- The descending order can be specified using the keyword **DESC.**
    - Eg. **b= ORDER a by age DESC;**
    - This command will arrange the rows in relation a in descending order on the basis of age. That is the oldest person's record will appear first.

10. **LIMIT:**

This command is used to restrict the number of rows in a relation.

- Eg. **b= LIMIT a 5;**
- This command will return the first 5 rows in relation a and store them in b.

11. **DISTINCT:**

This command is used to remove the duplicate rows in a relation.

- Eg **b= DISTINCT a;**

- This command will return all the unique rows from relation a and store them in relation b.

12. **GROUP BY:**

This command is used to group the rows of a relation on the basis of a column or multiple columns which is known as a group. All the rows of the relation containing the same value for group are grouped together in a bag.

ID	Name	Age	Salary
1	karan	23	818273
2	rohan	21	981021
3	vinay	23	828721
4	mohit	23	980721
5	rahul	22	918371
6	manish	21	652471
7	komal	22	721121
8	manoj	22	533371

Figure 1: Example database for Group by Relation

- E.g. for the above data, the command **b= GROUP data BY Age**; will return the following data:

```
{
    ((21),
        {
            (1,rohan,21,981021),
            (6,manish,21,652471)
        }
    ),
    ((22),
        {
```

```
                (5,rahul,22,918371),
                (7,komal,22,721121),
                (8,manoj,22,533371)
            }
        ),
        ((23),
            {
                (1,karan,23,818273),
                (3,vinay,23,828721),
                (4,mohit,23,980721)
            }
        )
    }
```

13. **FLATTEN:**

The FLATTEN command unnests the bags inside the tuples and creates new tuples. Consider the following relation:

data={(1,{(1,2),(3,4)})}

If Foreach data generate $0, Flatten($1) is applied to this relation, it will transform the data into {(1,1,2),(1,3,4)}.

Here, $1 column contains {(1,2),(3,4)}. After applying FLATTEN($1), $1 is changed to (1,2) and (3,4), that is the bag is removed. So when we generate $0 and FLATTEN($1), it adds the value in $0 to each tuple returned by FLATTEN($1) that is (1,1,2) and (1,3,4).

14. **TOKENIZE:**

The TOKENIZE command splits a string in a particular column on the basis of a delimiter. The delimiters that this command supports are double quotes [“ “], spaces [ ], parentheses [()], commas [,] and asterisks [*]. Consider a column with the value ‘David Miller’; if we apply Tokenize on this string, it will return ({(David),(Miller)}).

15. **UNION:**

The Union command merges the data of two or more relations. This is the same as the Union command in SQL, but there is one difference. Unlike SQL, the schema of the relations need not be identical. If the schema of one relation can be cast into the schema of the other relation, the union will be successful. Else, PIG will throw an exception.

Eg. **c=UNION a,b;** will merge the data of relation a and b and stores them in c.

16. **JOIN:**

It is used to combine the data of two or more relations into one relation. The joining of two relations is basically carried out on the basis of a common column or a group of columns. The resulting relation will then contain the columns from all the relations we have joined.

Eg. **a= load 'personal-details.txt' as (ID:int, Name:chararray, Age:int);**

b= load 'contact-numbers.txt' as (ID:int, PhoneNumber:double);

c= JOIN a by ID, b by ID;

Here, the relation **c** will have 5 columns namely **a::ID, a::Name, a::Age, b::ID, b::PhoneNumber.**

Here, the column ID is known as the key. For each match for column key in relation a, and b, a new tuple for relation c is created.

17. **Aggregation/Evaluation Functions:**

PIG provides a number of aggregation/evaluation functions. Some of them are:

- Concat(): To concatenate two or more columns.
- Min(): To find out the minimum value in a dataset.
- Max(): To find out the maximum value in a dataset.
- Floor(): To calculate the floor value of a float.
- SIZE(): To calculate the length of a string or the size of a field.
- LOWER(): Converts a string to lowercase letters.
- COUNT(): To count the number of tuples in a bag.

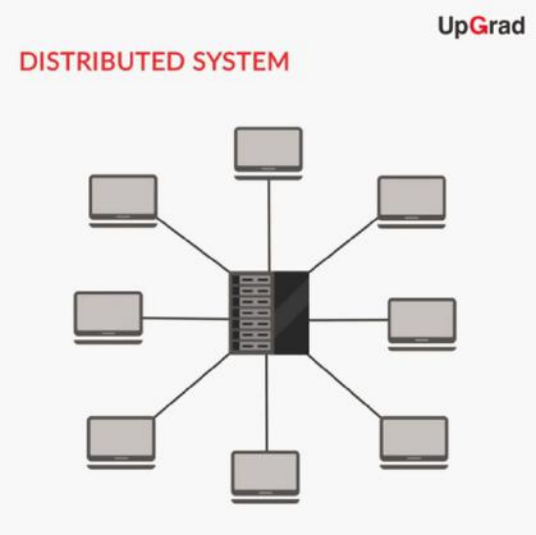
- AVG(): To calculate the average of the values of column in the bag.

Distributed Algorithm Design

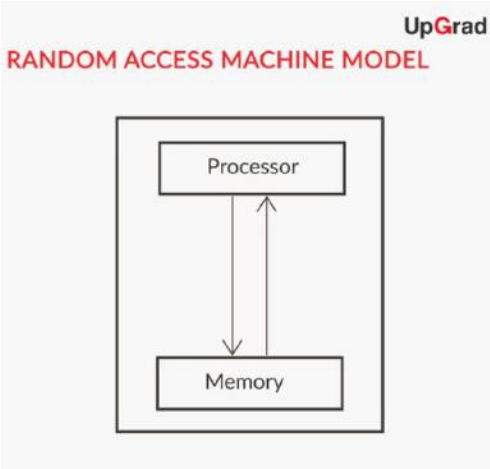
In the prior modules, you learnt about the top-down design and in particular, the divide and conquer technique for algorithm design. However, in those modules you assumed that all the data is stored and processed in a single machine. This assumption breaks down in the big data scenario, where a cluster of machines is required to solve problems. Hence, you required a new computational model that captures the parallelism in storage and processing.

Abstract Machine Tools

In this segment you learnt the basic abstract machine model for a single computer.



Previously you learnt about sequential machine that is typically referred to as a Random Access Machine model, where you have a processor and you have assumed that the prograre somehow has been loaded into it. An algorithm is about how to execute your prograre step by step. Further, while executing the steps, if you require data, you fetch it from memory and if you compute some results, you store it back in memory.



Now for designing algorithms for this kind of systems you need to understand the machine model at an abstract level. This model essentially consists of two pieces: there is a processor and a program, which sits inside the processor.

You can perform typical instructions in the processor such as:

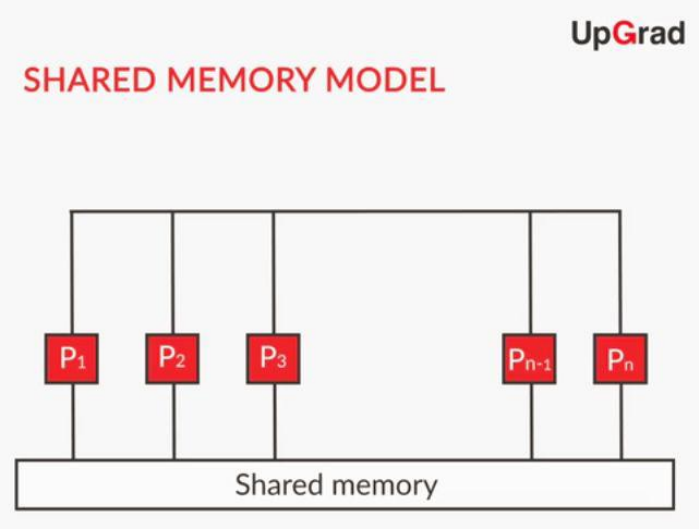
- Arithmetic and logic
- Load operations
- Store operations
- Jump or branch

In the parallel world, the machine model is more complicated than the sequential model. There are two kinds of machine models in the parallel world:

- Shared memory model
- Distributed memory model

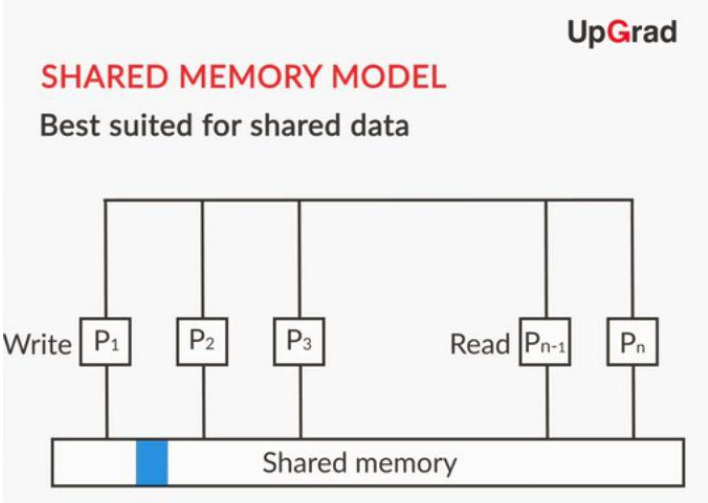
Shared Memory Model

In shared memory model, you have multiple processors, each can execute code in parallel and there is an interconnection network which allows processors to talk to each other if necessary as well as access memory. So there is a global memory(typically large in size) which is shared by all the processors and there is an interconnection network that allows the processors to access this memory.



In this model when you execute codes you are executing code in all the processors if possible. You write programs in such a way that you can execute the same code in all the processors and all of these processors can access the memory from different locations.

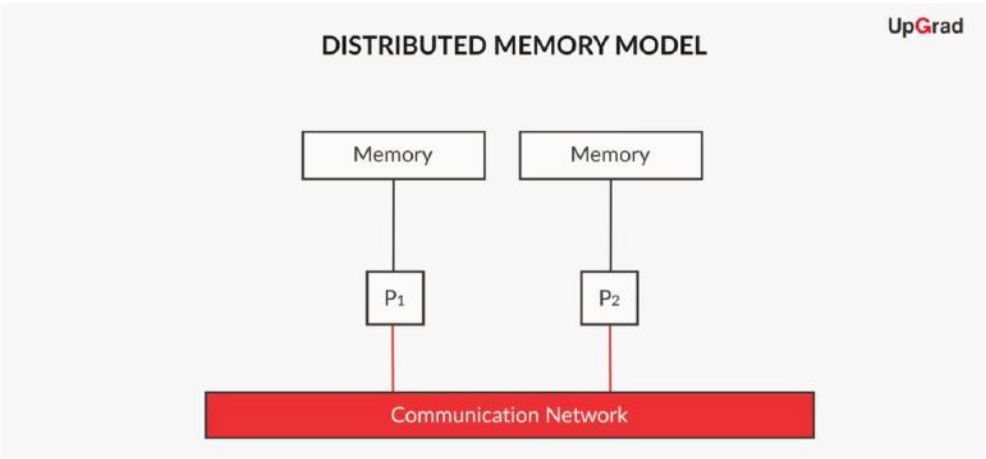
Note that this model requires a kind of programming where you need to share data. So if you don't have data that can be shared, then you will be using different parts of the memory for different processors to use. There is no advantage of doing that. If you have shared data where one processor reads the piece of data which has been written by another processor, then it is called shared data or shared access for data.



This model is very useful, for instance, in a multicore system, one core could be running a prograre which is updating a certain piece of data in memory and in the same memory you can access read from another processor. The same data is shared by the two threads for instance, which are running in the processors. The data can be stored in the shared memory and can be accessed by multiple processors.

Distributed Memory Model

The distributed memory model as opposed to the shared memory model is also a parallel machine model for which you will be designing algorithms. In a distributed memory model, each processor has a private memory, i.e. the memory available for each processor can be accessed only by the processor and typically every processor has its own private memory.

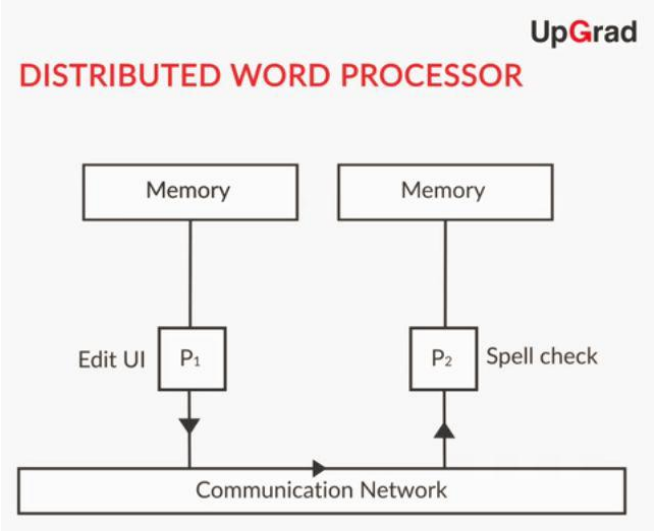


In this model, there is no notion of a global or shared memory. Hence, if a processor wants to communicate, they have to send messages to each other using a data network or a communication network. This communication network or data network is usually a separate

entity from processors. In your multiple processors or so called multiple nodes, each node may be running for instance its own operating system or other applications. So you can connect these nodes using a data network or a communication network. Then these processors or systems can talk to each other by sending messages.

In this machine model you have multiple processors running the same prograre where each processor will have an address space and each process will run in one processor. The address space of the process is mapped on to the memory address space of the processor and that is a private memory which other processors cannot access.

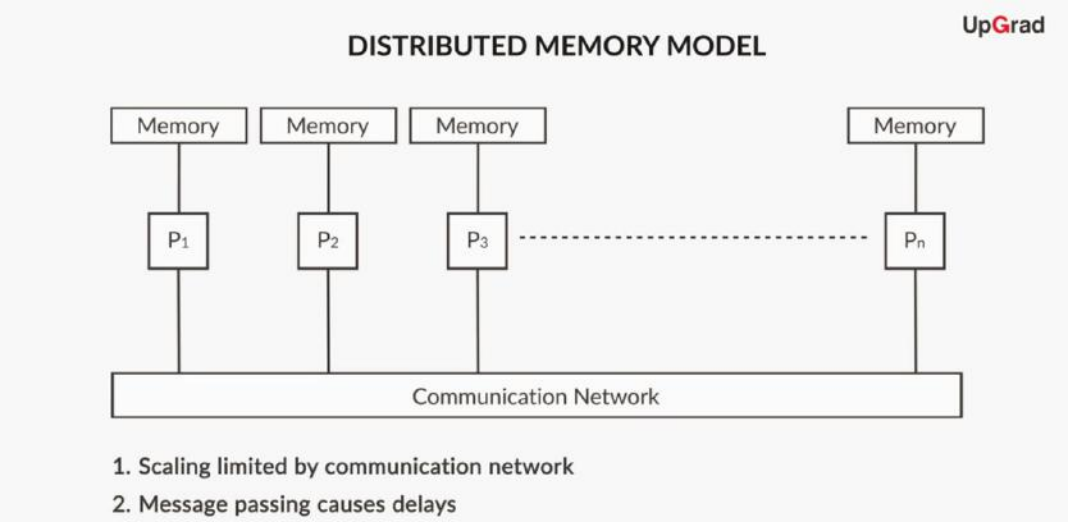
Therefore each process would be using its own personal data stored in the private memory of the processor where it is running. If you have to share data or you have to exchange data, that is done by message passing and the message passing happens over the communication network which is accessible by all the processors.



Consider an example of a text editor where you have to run two separate processes inside two separate threads and the data cannot be shared. If you want to run a spell checker separately as a process, you have to keep sending messages from the editing end to the spell checker end and this data has to be exchanged. The spell checker will do the spell check and then maybe do a correction and the correction is sent back.

In such a case there is no shared memory, so you cannot have shared data. So if you want to exchange data or you want to use some data which is processed by one node, it has to be sent as messages from one node or one system to another. This is the additional burden in this model but this model is useful because you can scale the system physically.

In this model every processor has a memory, therefore the amount of memory available for processing can increase along with the number of processors and it is possible to physically scale the system by adding more processors with more memory to the communication network.



So the limitation in the number of processors is due to the constraint regarding how many such nodes can be connected in the communication network for being able to communicate with each other. This model has its own flexibility in terms of scaling and in terms of adding more memory and more processors and this model uses a private memory instead of a shared memory.

So you can use this model for this kind of parallel programming but with the condition that whenever data has to be shared or cross-accessed, you have to use message passing for that. Message passing will cause delays compared to accessing things in shared memory. Typically for message passing over a network, the delay or the latency is in microseconds.

Pseudocode Convention

you use the following three concurrency keywords in our pseudocode to describe parallel or distributed algorithms. These are very similar to the keywords in the Cilk Plus programming language for multithreaded parallel computing.

- spawn
- sync

- parallel

The spawn keyword indicates that the instruction following it may be executed in parallel by a separate child node while the parent continues executing the next line of instruction.

The sync keyword indicates that the procedure must wait for all the spawned children to finish their tasks before proceeding to the next instruction. Usage of sync is extremely important as may be seen in the following parallel implementation of the Fibonaccyou generator. Note that you will prefix 'P-' to the pseudocode title to indicate its parallel implementation.

```
P-F(n)
if n<=2
        return 1
else x = spawn P-F(n-1)
        y = P-F(n-2)
        sync
        return x+y
```

Observe that the parallel implementation is exactly similar to the sequential implementation demonstrated in the previous module except for the concurrency keywords. The spawn keyword in the third line indicates that the instruction following it i.e. P-F(n-1) (computing the (n-1)st Fibonaccyou number) may be executed in parallel by a separate node while the parent node moves ahead and executes P-F(n-2). It's imperative to use the sync keyword in the subsequent line so that the sum x+y uses the intended values for x and y rather than some previously stored values.

Finally, the parallel keyword facilitates the use of parallel for-loops, i.e. each iteration of the loop is executed in parallel by a separate machine. Consider the problem of incrementing every element of an array by one.

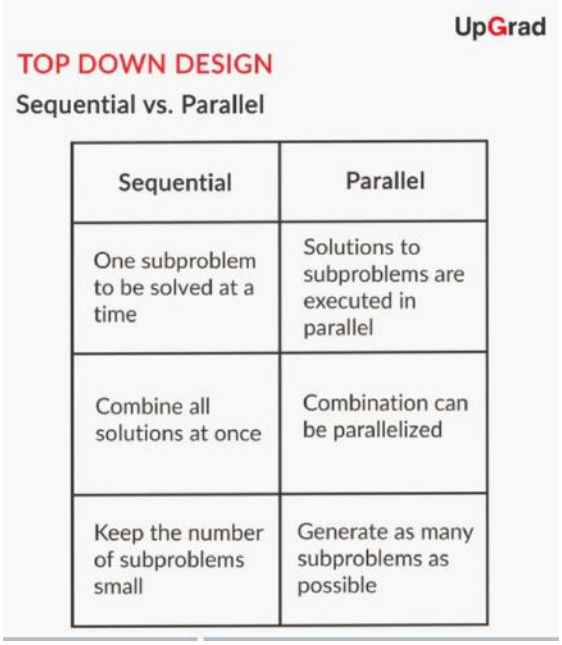
```
P-INCREMENT(A)
n=A.size
parallel for i=0 to n-1
        A[i] = A[i]+1
```

The parallel for indicates that each increment operation may be executed in parallel by a separate machine. Note that you could use parallel for in this problem since each increment operation is independent of the other.

Parallel Programming Models - I : The Data-parallel Model

In this segment you talked about how to do top-down design of algorithms as well as how to look at divide and conquer as a technique for designing algorithms in the context of parallel systems, in particular distributed systems. In our prograre you primarily looked at design for distributed systems, i.e. systems with distributed memory as opposed to shared memory systems.

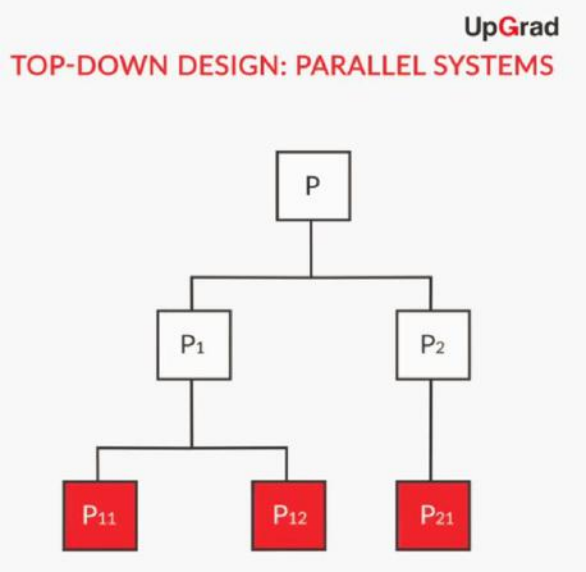
If you look at top-down design, you talked about dividing the problem into subproblems and solving those sub-problems one at a time and then combining the solutions, that's the typical approach for top-down design.



- The assumption in top-down design was that there was only one processor following the sequential execution model. Therefore, the subproblems have to be solved one at a time and you have to obtain all the solutions with the subproblem and then you can go on and combine the solutions. But in a parallel system, if you have multiple subproblems ,you can execute the solutions to the subproblems in parallel. Solutions to subproblems can be computed in parallel, i.e. at the same time.

- So if you have the solutions for the subproblems, you need to combine them and the combination may happen in one place or the combination itself can be parallelized. The technique for combining the solutions may itself be parallelized or you may need all the results in one place before you can do combine. This is one important differentiation between sequential world vs. top down design.
- In the top down design for sequential world one method was to keep the number of subproblems small in order to manage the complexity. If you have more than a few sub problems at one level then everytime you keep dividing the problem into smaller problems you will increase the number of subproblems that you have to manage. So to reduce complexity, the number of subproblems for a given problem should be kept small. But in a parallel system, you have multiple processors in which you can execute algorithms or execute code, which means, you should look to utilize those processors.
- Therefore you would want to generate as many subproblems as possible so that solution for each of these subproblems can be computed in parallel depending on the number of processor available. So typically the heuristic in top-down design is to keep the number of subproblems as many as possible i.e. divide the problem into as many subproblems as possible assuming that they can be executed independently.

TOP-DOWN DESIGN: PARALLEL SYSTEMS



If you look at a simple algorithm with problem p which gets divided into two problem p1 and p2 then you can execute p1 and p2 in parallel but executing p1 and p2 means dividing p1 further. so you will have p_{11} and p_{12} and let's say p2 has a sub problem p_{21} then you would want to

execute all these three subproblems in parallel. As you keep dividing you will keep generating subproblems. The idea is to generate as many sub problems that you can execute in parallel.

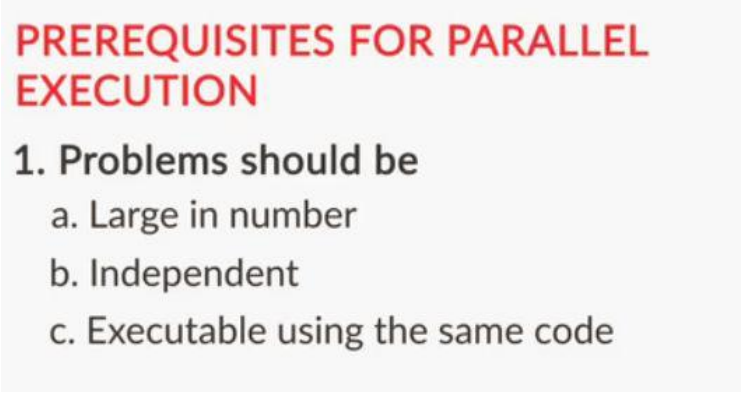
In general you also talked about divide and conquer as a technique where you divide the problem into subproblems and the structure of the subproblem is similar to the original problem or structure of one or more subproblems is similar to the original problem, that's a special case of top-down design.

To fit divide and conquer into the parallel world, when you design algorithms for the parallel worlds or for the distributed systems worlds you would need to do the following:

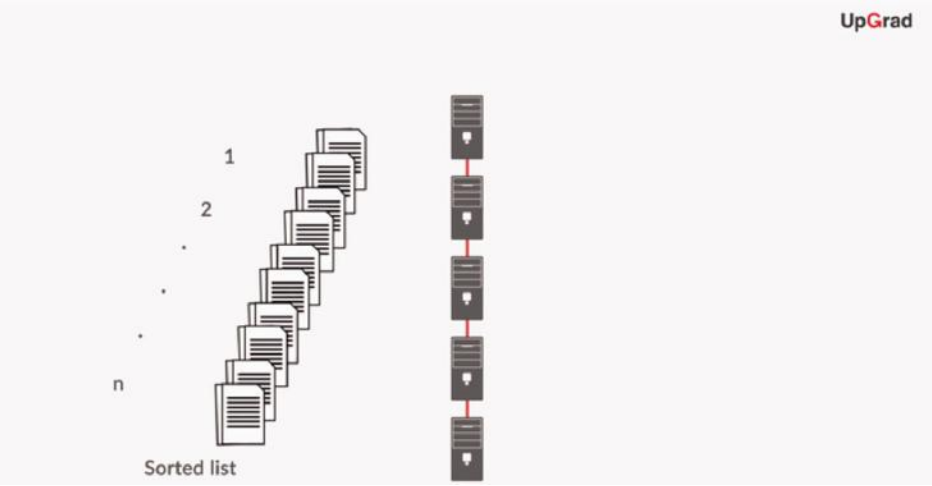
- Divide the problem into subproblems
    a. Sub problems must have similar structure
    b. Execute the same code on multiple machines

You can divide a problem into subproblems but if many of the subproblems are of the same structure then you can execute the same code on multiple machines in parallel, that's an advantage if you look at divide and conquer.

In general divide and conquer for the parallel world is similar to divide and conquer for the sequential world as an algorithm design technique. But having design the algorithm you would need to look for parallel execution.



So when you look for parallel execution you would want large number of problem which are independent. Also you would prefer to have the same code executed on multiple machines because then you can predict how long it takes to execute them in parallel.



If you give someone a sorted list and ask him to find a particular key, then you are going to divide it into sublists because it is a distributed system. The list itself is stored in a distributed fashion. The data is stored as sublists on different systems with different processors. Therefore, you are going to run the binary search algorithm on each processor on a sublist. So the advantages are:

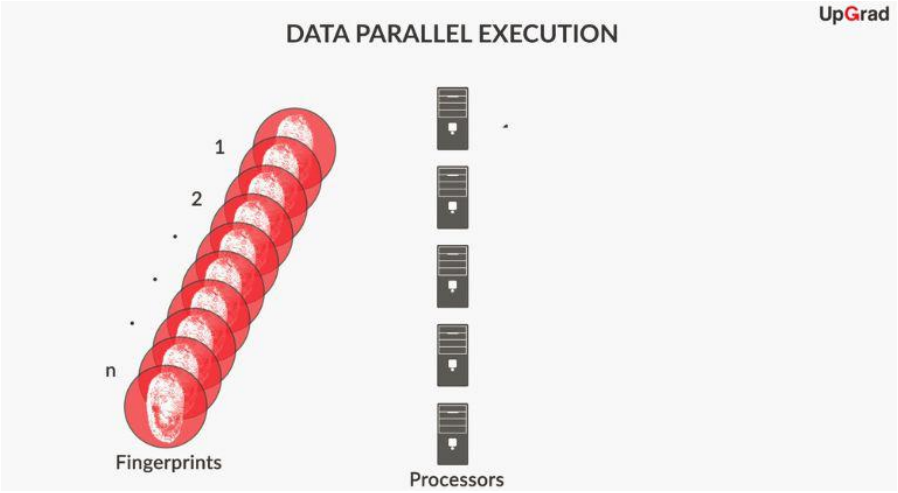
- The list is sorted, hence each sub list is sorted and you can run the binary search that will run efficiently on a sorted list.
- At the same time you have multiple processors that you want to utilize, therefore the problem of searching in a large list of 'n' items can be divided into 'p' subproblems where you have p processors and each subproblem is finding the same key that in a sub list of (n/p) items.

Now this subproblem can be solved by a sequential execution of binary search. So you can run binary search on p different processors at the same time. But in each processor you will be running it on a different sublist. This model is referred to as data-parallel execution where the same program or the same function is being executed on multiple machines.

- The data on which it's being executed is different, i.e. on different machines, you have different pieces of data but the same procedure or the same task is being executed on multiple machines.
- In data-parallel execution, there is no dependency between the processors that were executing in parallel, each is executing separately on different pieces of data. There is no data to be exchanged.
- Typically the amount of results that is to be collected is small.

Consider an example of matching fingerprints. So you are given a particular print which you have to match against a large collection of fingerprints available in a database because it is a

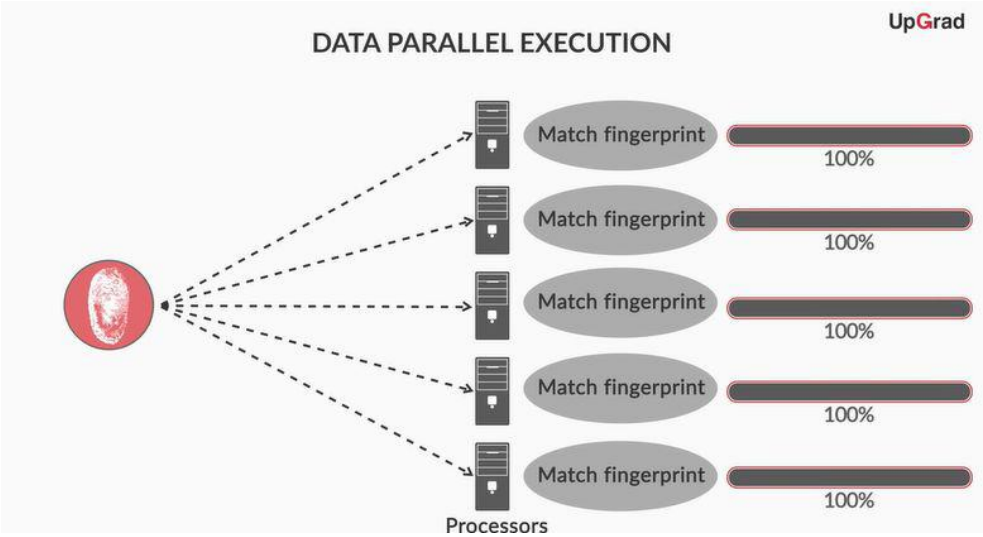
distributed system. Again you assume that the database of fingerprints is distributed evenly among the available machines.



If you have n different fingerprints in the database where n is large. So you are going to distribute this. Assume that the database is distributed among p processors and each processor has n/p fingerprints.

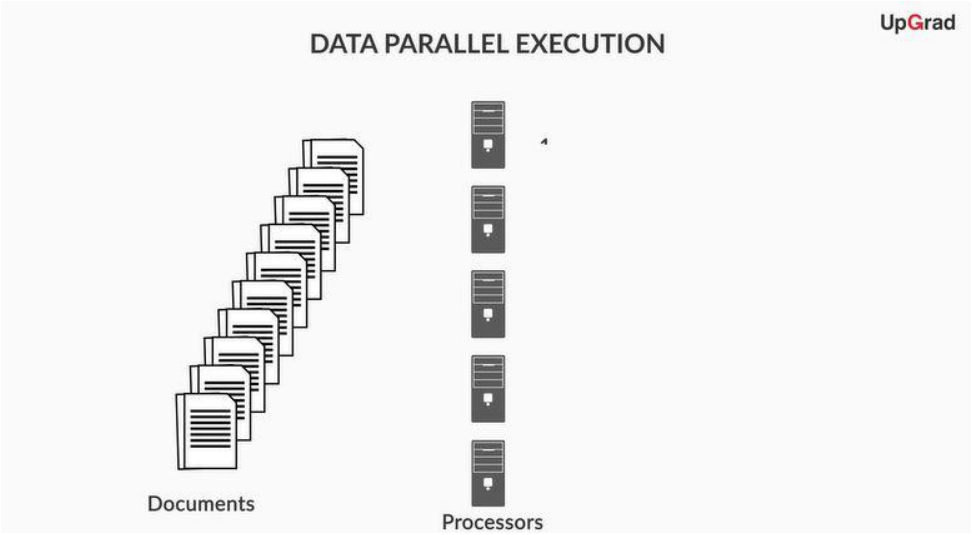
Now all you have to do is to send this single fingerprint that you have to match to all the processors and then each processor can run the matching algorithms take this single finger print and match it with each of the finger prints available in the database that is available with that processor.

In that database each fingerprint will be compared with the given fingerprint. Now this task can be executed in parallel because fingerprint matching is an independent process: for matching Fingerprint F with two different fingerprints D1 and D2 in the database, they can be done independently.

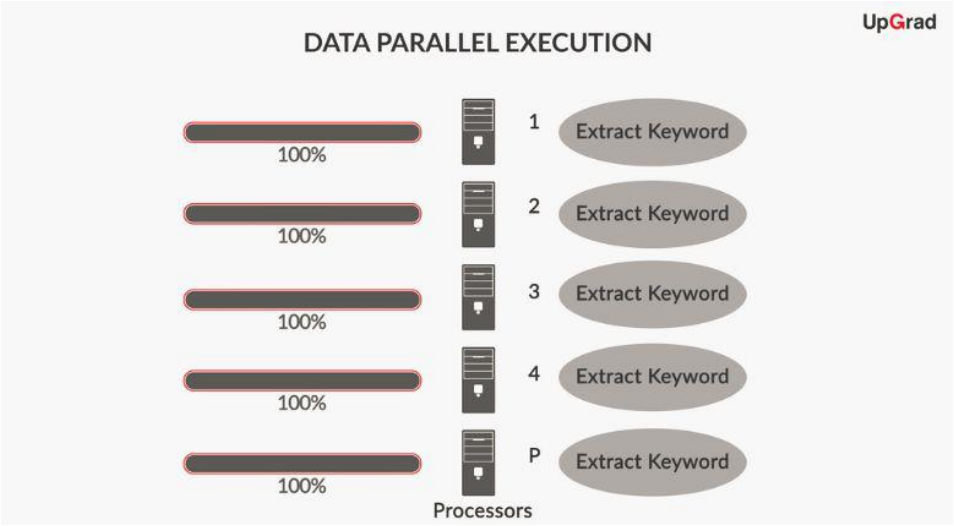


So the number of fingerprints you have to match in each processor is also roughly the same as you have divided the database evenly. So, given this, you can predict that all these processors would finish roughly at the same time. That means, you have utilized the processors effectively all the time and the total amount of time taken for all the fingerprint matching have been reduced by a factor of P because you are using P processors.

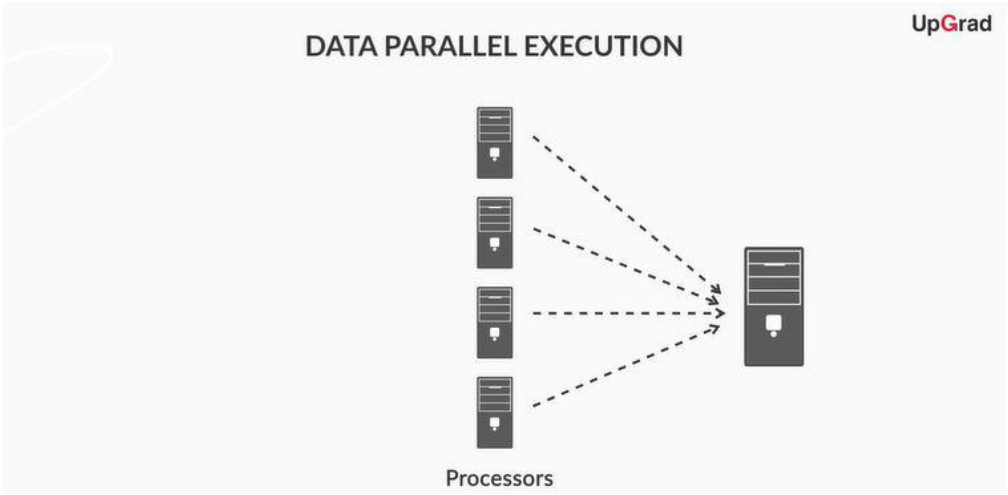
Similarly, you can look at another example, you have a large collection of documents which you have downloaded from the web and my goal is to extract keywords from all of these documents and collect them into an index. So, if you look at the first step of this, extracting keyword from the document is the same thing that as what you have done for multiple documents that you have collected.



So, you will extract keywords from one document, then go to the keyword extraction from the next document and so on.



So, if you have n documents, and you have P processors, then you would distribute the database of n document into n by P documents each and put them on different processes and your task of extraction from multiple documents will run on each of these P processors in parallel again. They will all finish roughly at the same time.



Once they are all finished, you can collect all the keywords from all processors and put them in one location, let's say in processor p. If you want to have a single index which has to be collected, you will collect it and put it there.

So this is again an example of data-parallel execution. In the first step, all processors are doing the same thing and doing it in parallel. But they are working on different parts of the data or on different data sets. So this is another example of data-parallel execution of the program.

In summary, in the data-parallel model, a given problem is split into several smaller instances and distributed among the available machines. Each machine independently processes the

subproblem assigned to it. After all the machines finish processing their instances, the results are aggregated if required.

The data-parallel model is most suitable for problems that can be broken down into a large number of independent subproblems, e.g. Keyword extraction from a large database of documents.

Parallel Programming Models - II : The Tree-parallel Model

We talked about data-parallel execution in a distributed system where the processors are executing the same process, but running in multiple nodes operating on different pieces of data.

Now it is not always possible to divide your program in parallel such that the processors run the same process. So in some situations, you have to divide your data before you can get into a scenario in which you will use data parallel model of execution. But in some cases, there are dependencies that you have to work around in the sense that when you have to divide the problem into subproblems that division does not necessarily give equal sized subproblems. But the subproblems are of the same structure of the original problem. In which case, you can run these two subproblems in parallel but the subproblems may take a different amount of time. So there is a dependency in terms of saying that, how much time each processor will take, i.e. there's a time dependency. There is no data dependency here. So these two subproblems may take different time to complete. So you have to wait for one subproblem to finish. If you consider the overall time taken or the processor utilization, so this may be a scenario where you may want to subdivide the problem with larger size into further subproblem so that you can execute those in parallel and so on.

If you take the example of Quicksort, you have a divide and conquer solution. You ran a partition process which takes a pivot element P and divides the given list into two sublists. All items less than the pivot element into one sublist, all item greater than the pivot element into another sublist and then these two sublists can be sorted independently of each other and the sorting process is the same problem, which has the same structure as the original problem and you can sort these things independently. In the divide phase of Quicksort, you run the partition algorithm.

UpGrad

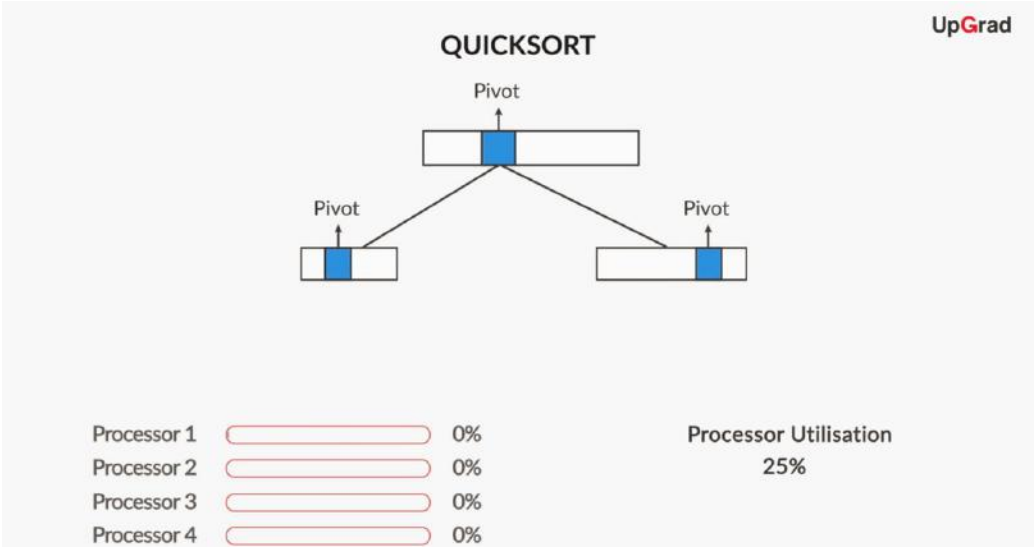
QUICKSORT ALGORITHM

QUICKSORT(A, lo, hi)	PARTITION(A, lo, hi)
1. if lo<hi	1. r = random(lo, hi)
2. p = partition(A, lo, hi)	2. pivot = A[r]
3. quicksort(A, lo, p-1)	3. i = lo
4. quicksort(A, p+1, hi)	4. for j=lo to r-1
5. else return	5. if A[j] ≤pivot
	6. exchange A[i] with A[j]
	7. i=i+1
	8. exchange A[i] with A[r]
	9. return i

Once you have run the partition algorithm then you can run two subproblems in parallel but the two subproblems need not be of the same size. Partition does not always partition the sublist into even sublists and hence the size of the two sublists may differ and therefore when you look at running these two in parallel, they may take different amounts of time.

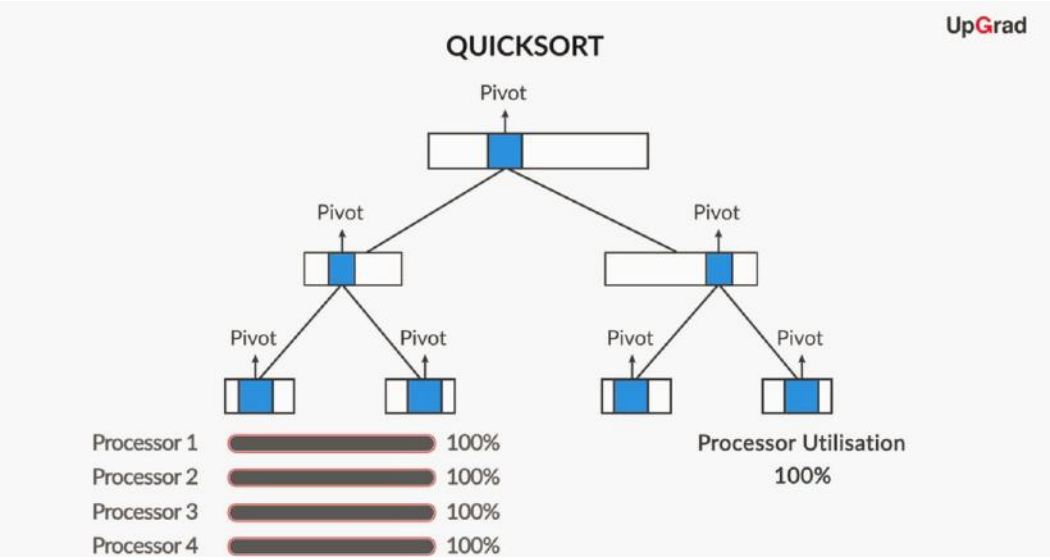
So you can take the sub list and you are going to divide them further which will happen in divide and conquer anyway. So you have to partition those sublists and then you have to divide them into two sublists and run those two sublist sorting problems in parallel. So if you look at it this way, in the earlier model you said you are going to run the same process at the same time. Whereas in this Quicksort model, it's going to happen that if you run this way, then you have to wait for other sublists to finish, so that's not appropriate. In Quicksort you will take the problem and divide them into subproblems. You will use two processors to run these two subproblems but these two processors will run independently anyway. Therefore, when they are done with the partitioning one of them will be done faster maybe because the size of its list was smaller, so that will generate two more sublists and and those two subproblems can now be started independently from the previous sublist.

Let's assume that the left sublist is smaller, therefore that finishes partitioning earlier. So you will split it into two further sublists and they will start running while the previous partitioning is still going on for the right sublist and whenever that finishes then that will have two sub-lists and those two sub-lists can be run in parallel now. The net effect is that you are running multiple processors but at any particular time you don't know how many processors are running.

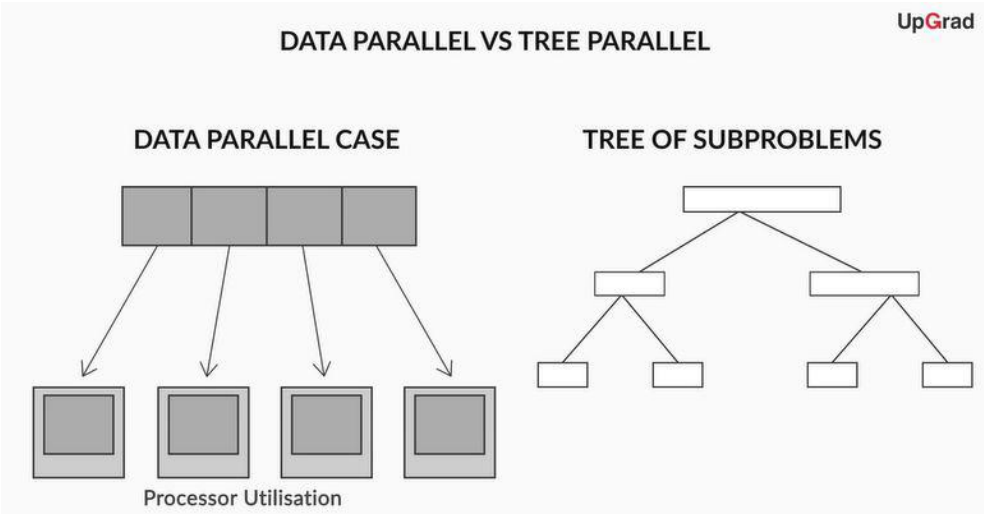


In some cases you may be running one processor or in some cases you will be running two processors, later the two may become three because one of the sublists got partitioned early and those two partitions can be further processed in separate processors but the second sublist you had in the top that didn't get partitioned early so it is still running.

So you have three processors running now. When that sublist also finishes partitioning, you have got two processors. You can use two more processors and run four processors in parallel. So the number of processors that are getting utilized at any point in time will vary because whatever is the number of sublists that are ready for partitioning those will be consumed by processors and those many processors would be running different processes.



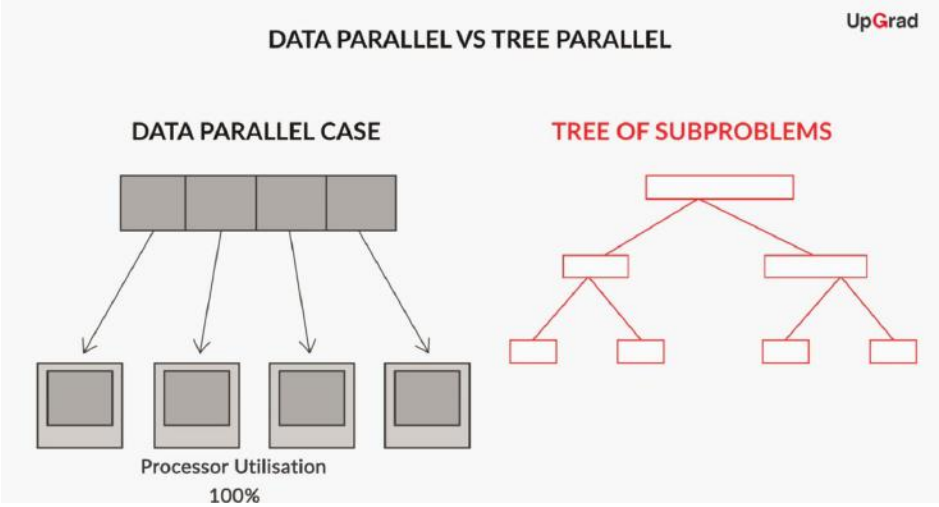
All the processors are doing the same thing, they are all running the partition algorithm at any point. Each processor is simply running the partitioning algorithm but the lists that are being partitioned are not of the same size and the lists are getting generated at different times. In this case what is happening is that you are actually generating a tree of subproblems as is the case anyway but the tree of different subproblems is getting assigned to processors at different times. In the data-parallel case, you assign all the processors together to different parts of the data at the same time. If the size estimation was good, i.e. approximately even, they will all finish at the same time.



So the utilization is exactly parallel utilization, i.e. hundred percent during that period whereas in this case what is happening is you're generating your sub problems at different points of

time. One of the nodes in the tree will get assigned to a processor that is available and at a particular point you may have different processors running at different levels of the tree.

If you don't have enough processes at one point in time, say you have only four available, then only four processors are getting utilized. If it keeps on dividing, eventually you will get small enough sublists, several of them, and they will occupy individual processors.



Therefore you will have a large number of processors all of them doing the same partitioning algorithm. So in this model what you got is known as **tree parallelism** where multiple nodes in the tree are being processed in different points of time depending on the number of processors available and depending on the number of processes that have been generated. So in tree parallelism, in general, you keep dividing the problem into subproblems and those sub problems are assigned to processors and those processes will be partitioning them.

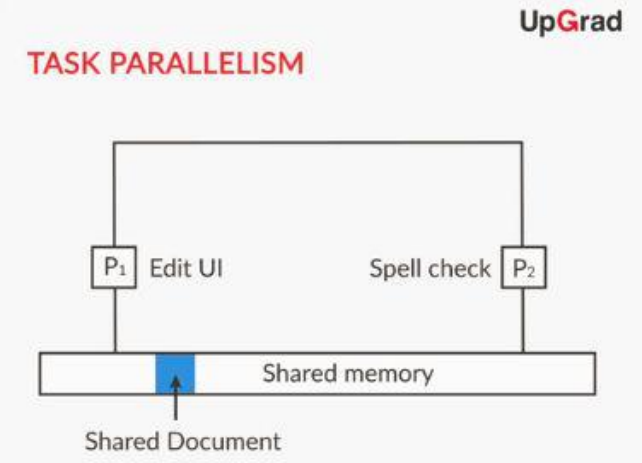
In summary, the tree-parallel model, as the name indicates, accounts for dependencies between a problem and its subproblems. In this model, each subproblem is dynamically allocated to an idle machine, thereby optimizing the processor utilisation, e.g. in parallel quicksort, each array is partitioned and the resulting subarrays are partitioned in parallel. This process continues resulting in a tree of subproblems.

Parallel Programming Models - III : The Task-parallel Model

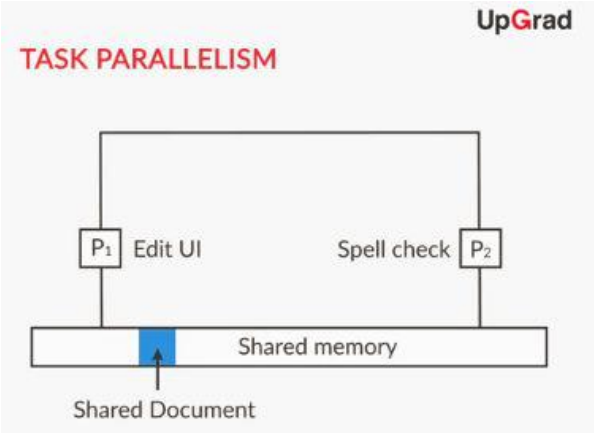
So you talked about data parallelism and tree parallelism. In the examples you looked at, the parallel processing is happening in such a way that the processes are doing the same thing on different pieces of data.

So you are running multiple processes, they're all running parallely here on different machines but each process is doing the same thing as the other processors, all of them are processing the same kind of task but on different pieces of data. This may or may not always happen there will be problems where you have different pieces of data or you have to do different tasks and both may also happen, i.e. you have to do different tasks on different pieces of data or you have to do different tasks on the same piece of data.

So if this is the situation, then you talk about task-parallelism. So you looked at couple of examples, one is the example that you have looked at before where you have a word processing program in which while you are editing and you want to spell-check in parallel.



So you saw this in the case of shared memory parallelism where you said you can run a separate thread for spell checking and the main thread of editing and the user interface will run in one thread and the spell checking part will run in another thread. You can look at this as an example of task parallelism where you have two different tasks and these two tasks can happen in parallel although in this particular example the data is being shared, they are working on the same piece of data.



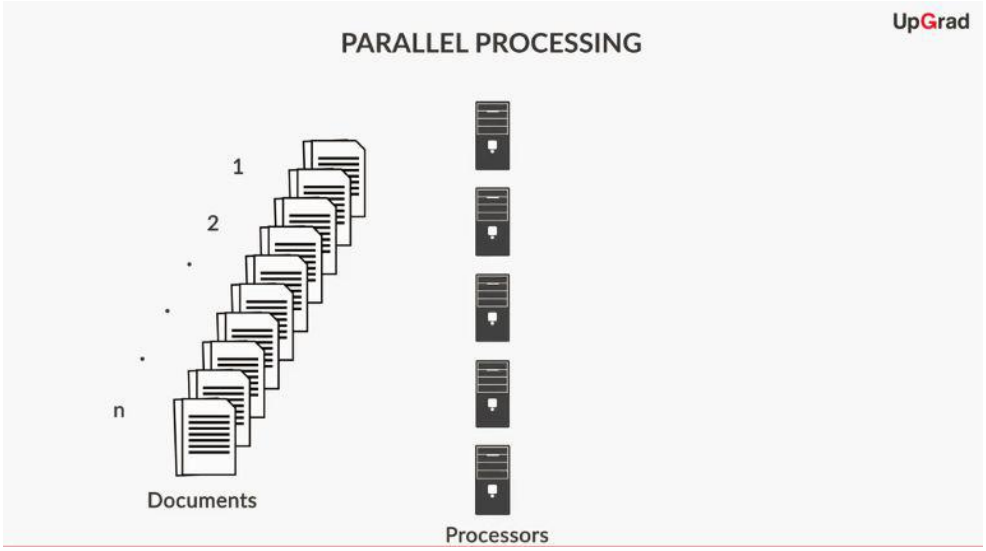
Let us look at another example of task-parallelism. Let's revisit the example where you were doing document indexing and you had to extract keywords from a document.

Suppose that you change the problem slightly. From the collection of documents that you have downloaded from the web, you need to extract keywords and also extract images. But the algorithm for extracting keywords will not work for images and vice-versa. These are two different techniques or algorithms.

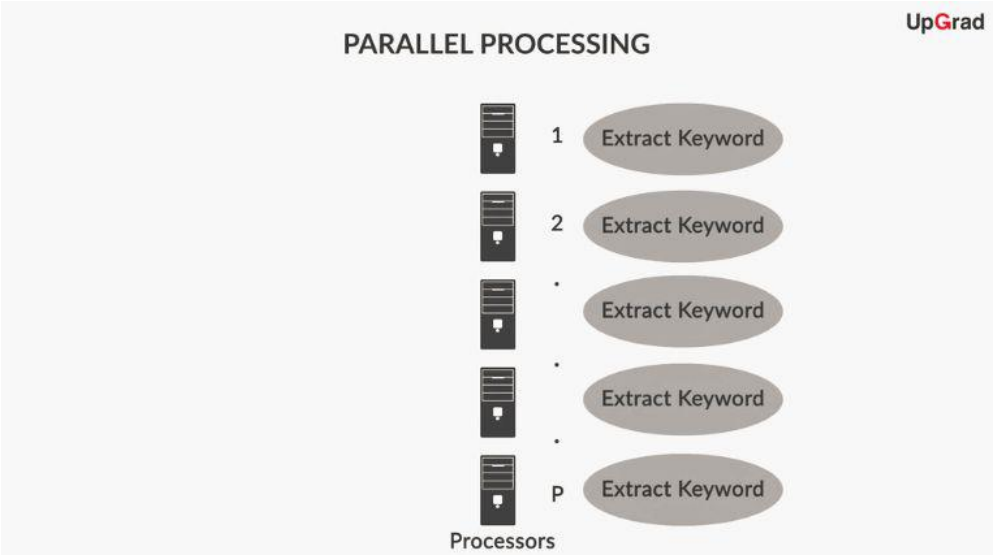
You have downloaded the documents. In one case you are going to extract keywords, i.e. the output is keywords and in another case you are going to extract images, i.e. the output is images. Neither of these algorithms will affect the other, so extracting images is not going to affect what the keywords are and extracting keywords is not going to affect what the images output is, so you can do these things independently.

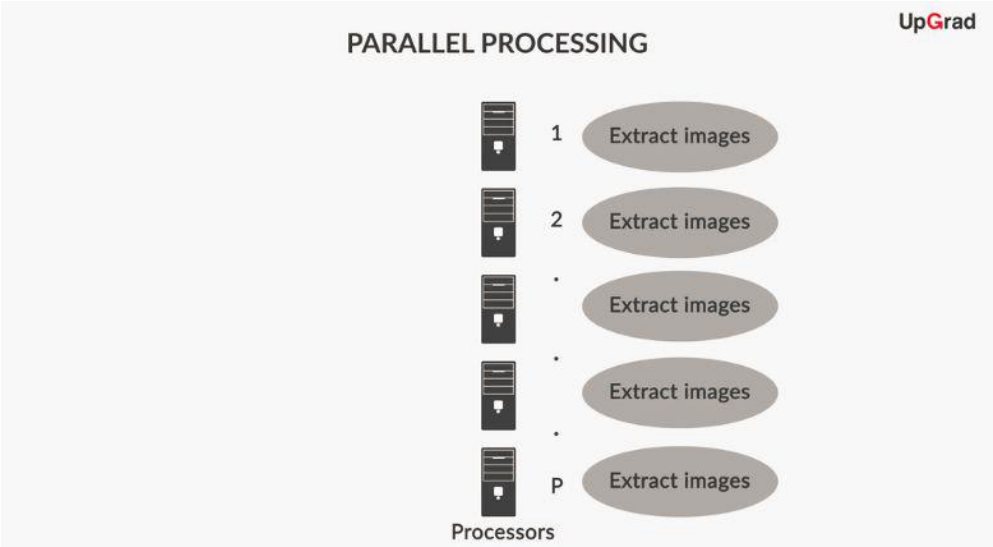
So although there is data which appears to be shared, it is not actually shared if you make copies. If you are willing to make copies, then you can perform these two tasks separately.

If you can perform these two tasks separately, you can perform them in parallel. So for every document it is possible for me to run a separate process. But you don't want to run a separate process for each document. So you are going to run a process for the collection of documents.

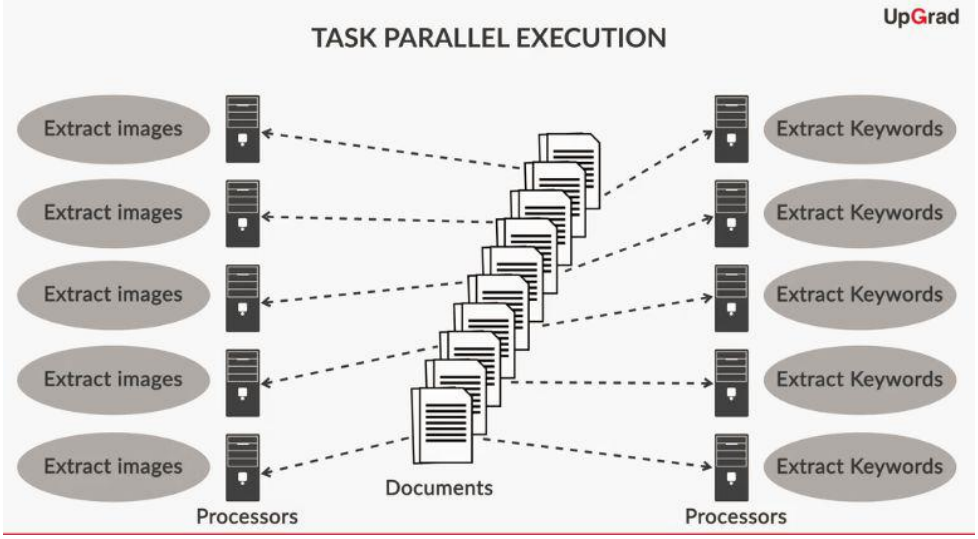


If you have one copy of all documents and they are distributed you can run on that copy all the keyword extraction and maybe you can follow that up with the image extraction algorithms. This will take time, one set of one amount of time for keyword extraction another amount of time for image extraction.





But if you want to do these two tasks in parallel, you can do that also. You make copies of this data. All my keyword extraction algorithms are running in parallel on P processors. Suppose you have additional P processors you can run all the image extraction algorithms also in parallel, so these two tasks will run in parallel. The time taken will be the sum of these two.



Performance Analysis of Parallel Programming Models

In this segment, you learnt about the performance of distributed algorithms for which you talked about the design in the previous segment. The goal in using parallel or distributed systems is to put in more processors and gain advantage in terms of time taken for their execution.

The reason we're putting in more process, i.e. putting in more cost is to get a benefit and that benefit is that the running time will be reduced. So our goal is to reduce the time taken by an algorithm when you are designing a parallel or distributed algorithm. The approach is to ensure that all the processors are utilized fully and this can be achieved if you utilize the processors uniformly.

So if you keep the processors engaged, typically the performance will turn out to be good that's a thumb rule. Now the ideal speed-up you can expect if you put in P processors is P, speed-up is referred to as time sequential by time parallel. The time taken for a sequential algorithm divided by the time taken for the parallel algorithm and if you put in P processors, the ideal speed-up you want is P, that means you put in a cost corresponding to P processors and the benefit you want is proportional to P, this is an expectation.

Now ideal case is rarely achieved there are always obstructions to achieving the ideal scenario. The primary reason for this are dependencies in the computation, that is, you cannot execute all of them in parallel. One computation depends on the other, so if the first computation is not finished you can't go to the second computation.

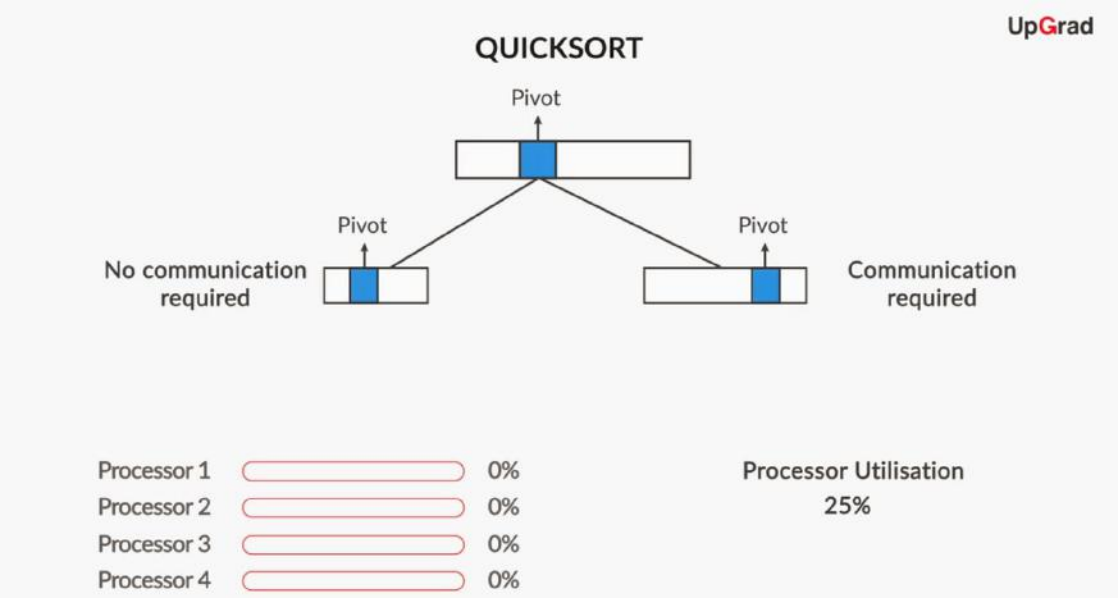
There is something that is preventing me from executing things in parallel that is dependency. There are two kinds of dependencies that may naturally arise. One is the presence of sequential parts in an algorithm. Suppose that you have to do small steps and you have steps $s_1, s_2, \ldots, s_k$ where s_j cannot proceed before s_{j-1} for all $1 \leq j \leq k$ and there are several substeps.

In this case, all the k steps are to be executed only on one processor and therefore the time taken cannot be reduced below the sequential time required for k processes. The other situation is where you have to communicate between steps, which is a data dependency. In the first case, you saw when you said there is a sequential part of it, that's control dependency. When the execution cannot proceed from one step to another before the previous step completes, that's a control dependency. But you also can have data dependency where whatever is computed in one step that data has to be passed on to the second step.

So if you run these things in parallel, the steps are independent in the sense that you can perform them independently but, it doesn't make sense to do the computation without the

data that is obtained from the previous step. So communication between steps is required if you run them in parallel.

For example, in our quicksort algorithm which you modified for distributed system, after every partition step you have to send the sublists to new processors so that the new processors can execute those sublists.



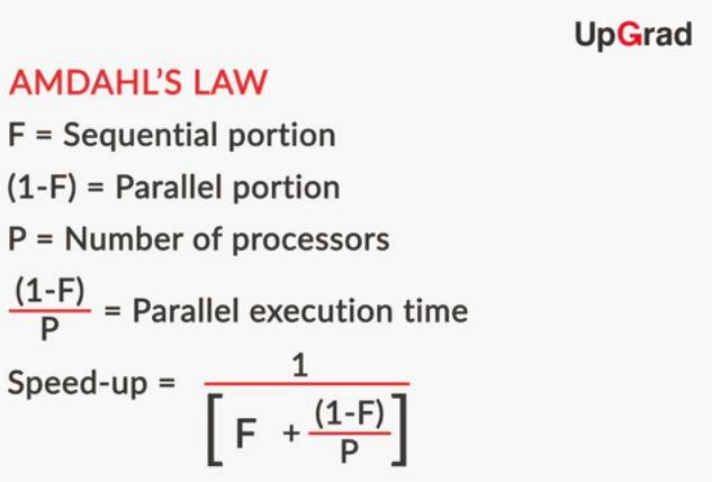
You can also think of that as saying that you divide the list into two sublists this processor itself will execute one of the sublists, it will take up one of the sublists but still you have one more sub list if you want to execute that in parallel you have to send it to another processor.

So there is a communication step in the case of a distributed system that has to be achieved by passing messages and that's going to have latency. Therefore that latency will be part of the time taken for computing. So the total time taken will be increased by the communication that is required.

So two forms of dependencies can manifest, one is there is a control dependency between steps 1 and 2 or there is a sequential data dependency between steps 1 and 2. The data dependency will manifest in the form of communication cost typically. So given these two limitations your performance will not always be ideal so you need to expect a performance which is not ideal. i.e. if you put in P processors, the ideal speed-up is P but because of the dependencies and the communication that is required you may not achieve ideal speed-up.

So there is a simple formula which allows you to estimate how much is the expected performance or what is the performance you can expect. so this is referred to as Amdahl's law which says "*speed-up is dependent on the sequential portion*".

Suppose you have a sequential portion F, i.e. F is a fraction of the program that has to be executed sequentially, that means (1 - F) is the portion that can be executed in parallel. Now if you can execute it in parallel and you have P processors and you achieve ideal speed-up for that P processors, so you have F as the sequential part of the time and (1 - F) divided by P as the parallel execution part of the time. You can combine these together and then say that this is the time taken using the parallel technique. So time sequential by time parallel with P processors would turn out to be $\frac{1}{F+\frac{1-F}{P}}$.



Now this factor is critical to understand because it quickly allows you to estimate what is the maximum speed-up you can get. Now one has to note that even a small value of F will limit the amount of parallelism you can get.

UpGrad

AMDAHL'S LAW

F = Sequential portion

(1-F) = Parallel portion

P = Number of processors

$\frac{(1-F)}{P}$ = Parallel execution time

$$\text{Speed-up} = \frac{1}{\left[0.1 + 0.0009\right]}$$

Suppose P = 100

UpGrad

AMDAHL'S LAW

F = Sequential portion

(1-F) = Parallel portion

P = Number of processors

$\frac{(1-F)}{P}$ = Parallel execution time

$$\text{Speed-up} = \frac{1}{0.1009} < 10$$

Suppose P = 100

UpGrad

AMDAHL'S LAW

F = Sequential portion

(1-F) = Parallel portion

P = Number of processors

$\frac{(1-F)}{P}$ = Parallel execution time

$$\text{Speed-up} = \frac{1}{\left[0.1 + \frac{0.9}{\infty}\right]}$$

If 10% of your program is sequential, then as you increase the number of processors, you are reducing the time taken for the parallel parts but the time taken for the sequential part remains

0.1. So the total time taken will not go below or 0.1 that means the speed-up you can achieve at the maximum is 10.

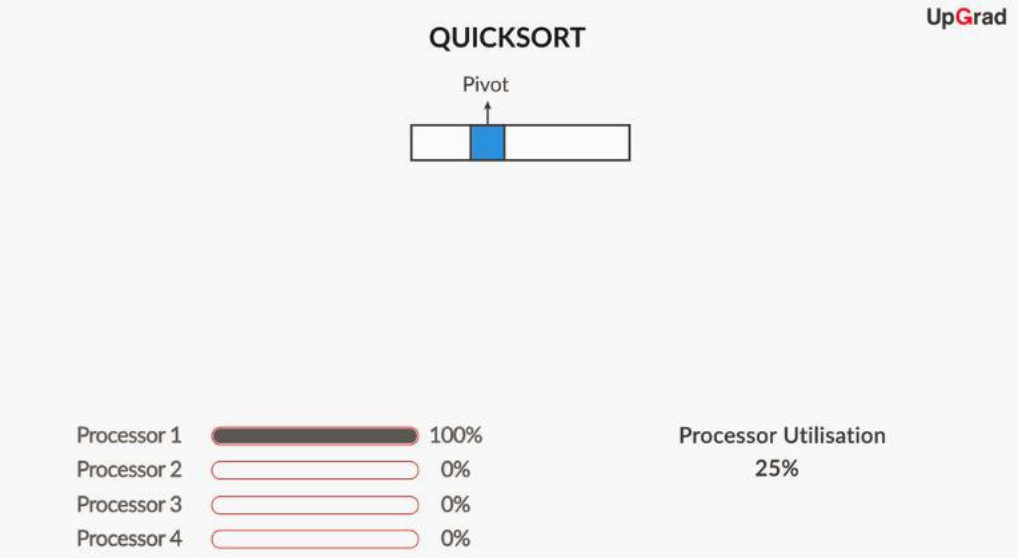
This means if you have 10% sequential portion you will get a speed-up of maximum 10 irrespective of number of processors you put in. If you get 50% for instance of sequential part then you're not going to get a speed-up of more than two irrespective of the number of processors that you put in.

So this means that when you write your algorithm, you should think through and ensure that any sequential part is minimized and the way to minimize sequential part is to observe the dependency and try to eliminate the dependency, so that the sequential part is minimized in-turn and your expected speed-up is increased.

QuickSort

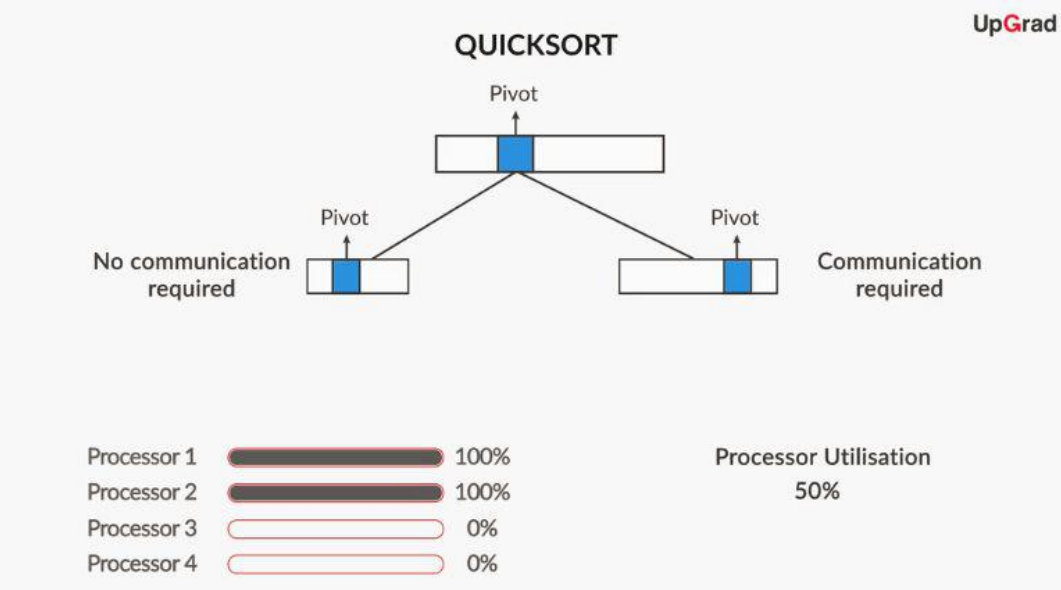
If you look at the Quicksort example for instance the processor utilization in this case is low in the beginning. This is because there is one partition algorithm running on the entire list you can use only one processor.

This means a dependency situation because unless you do the partitions you cannot run them in parallel and the two sub-lists or sorting the two sublists cannot be run in parallel until the partitioning is done. So there is this dependency, once a partition gives two sublists you can run those two sublists you can sort those two sub lists in parallel but you cannot go further until each of those sublists is partitioned.

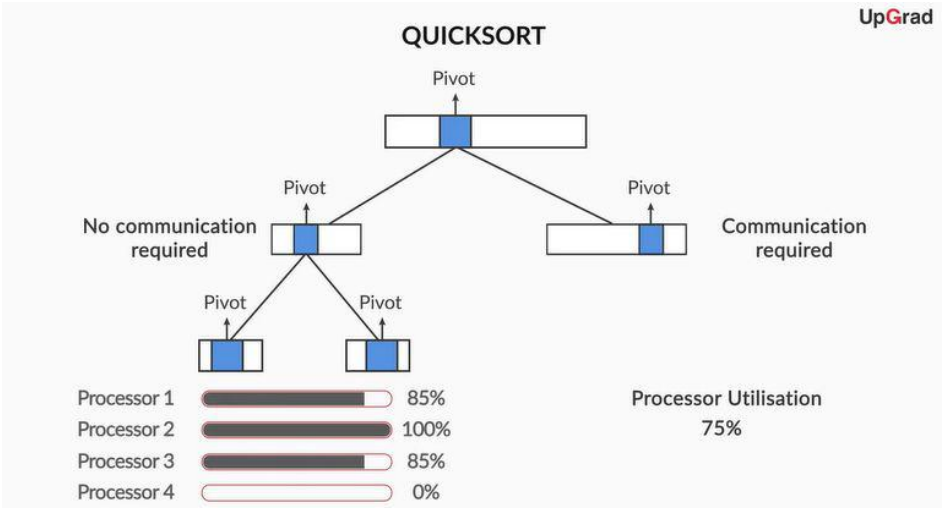


So if one of those sublists is partitioned, you can go further with two more processors. But the other sub list may still be running sequentially. So if you look at quicksort, the processor utilization in the beginning with one large list is very low, as you keep on subdividing the list, as

you keep on running partition algorithms and generate more and more sublists, you will be able to run more parallel processes.



So the utilization increases towards the higher end of partitioning or towards the lower end of the sub list size. As the number of sublists increases, you can run more and more partition processes in parallel, but this means for significant part, your algorithm is running sequentially or your algorithm is running with small number of processors and the utilization is low so these sequential parts will affect the performance.



So in this case you will not get an expected speed-up which is proportional to P, the number of processors that you are utilizing.

To summarize, in data-parallel situation where you can execute the same task in multiple processors and they are all executing independently on different pieces of data, the speed-up is P where P is the number of processors since all the tasks are executing independently.

If you have n items to be processed, each processor is processing n/P items so the time taken by each processor is n by P. All the processors finish roughly at the same time and hence, the total time taken has been reduced by a factor of P, i.e. the speed-up is P.

But if you have task-parallelism on the other hand, the task-parallel system is limited by the largest task. If you take the document indexing example, keyword extraction is a fairly simple process but image extraction could be a complex process. So if you run two different tasks in parallel, one for keyword extraction and one for image extraction, then it may turn out that the image extraction process is taking much more time than the keyword extraction.

If this happens the total time taken is going to be roughly proportional to the image extraction time, that is close to the total time taken, let's say the image extraction 95 percent of the total time taken between image extraction and keyword extraction, that means the parallel time is going to be 95 percent of the sequential time.

So the speed-up that you obtain is 1 over 0.95 which is not very large so parallel tasks parallel activities will not give you good performance unless the tasks are divided into equal parts. so how do you address the situation?

In this particular example you need to see if you are doing keyword extraction from one document and and image extraction is taking let's say ten times more time than the keyword extraction process, you need to ask whether you can speed up the image extraction process, for instance, you can you run the image extraction process in parallel.

If you run the image extraction in parallel, then if you have a task of extracting documents and you will put one processor to extract keywords from one document and at the same time you will put ten processors to extract images, maybe that will speed up the image extraction time and the tasks will be evenly divided in terms of performance. Then you can hope to achieve good processor utilization and therefore good speed-up.