**HDFS**

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault tolerant and designed using low-cost hardware. HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

**Features of HDFS**

* It is suitable for the distributed storage and processing.
* Hadoop provides a command interface to interact with HDFS.
* The built-in servers of namenode and datanode help users to easily check the status of cluster.
* Streaming access to file system data.
* HDFS provides file permissions and authentication.

HDFS follows the master-slave architecture and it has the following elements.

**Namenode**

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks:

* Manages the file system namespace.
* Regulates client’s access to files.
* It also executes file system operations such as renaming, closing, and opening files and directories.

**Datanode**

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.

* Datanodes perform read-write operations on the file systems, as per client request.
* They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

**Block**

Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a Block. The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

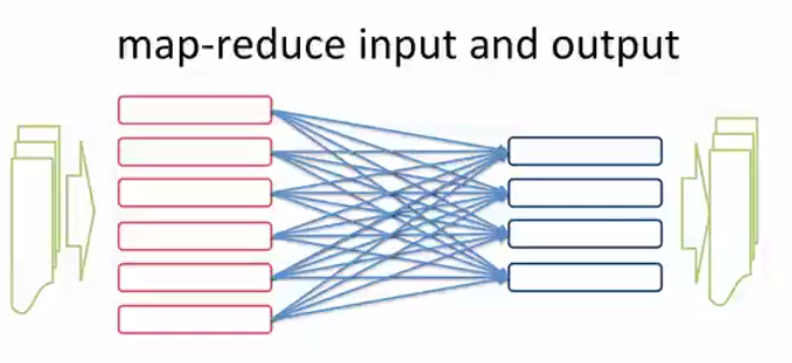
**Goals of HDFS**

* **Fault detection and recovery** : Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.
* **Huge datasets** : HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.
* **Hardware at data** : A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

**Why HDFS works very well with Big Data?**

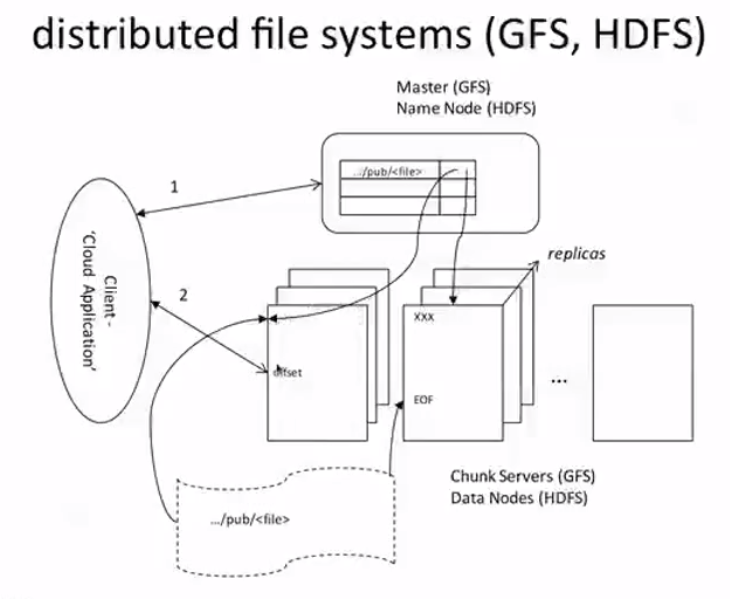
* HDFS uses the MapReduce method for access to data which is very fast
* It follows a data coherency model that is simple to implement still highly robust and scalable
* Compatible with any kind commodity hardware and operating system processor.
* Economy is been achieved by distributing data and processing on clusters with parallel nodes
* Data is always safe as it is automatically saved in multiple locations for safe secure.
* It provides a JAVA API’s and C language is on the top priority.
* It is easily accessible using a web browser making it highly utilitarian.

Map-reduce reads and writes fresh data. As we've seen before, it's a batch process. The question is, where does it read this data from, and where does it write the output to? Even if map-reduces is a parallel computing paradigm, what about the data read and write? Does that form a potential bottleneck?



The solution is that the data itself needs to lie in a distributed format, whether it's a distributed file system or a database so that we can support parallel reading and writing. So, that more than one processor is active reading or writing a large file.

Otherwise, input and output itself will become a bottleneck for the map-reduce parallel computing program. We know that processing node failures are recovered without letting the overall map reduce job fail. But, what if the nodes on which data is stored failed, they also need to be fall tolerant. To solve both these problems, distributed file systems were developed primarily first by Google, and then Yahoo



This diagram illustrates the architecture of the, distributed file systems, both the Google file system as well as the, Hadoop Distributed File System which is Open Source now, and is essentially the foundation of what is big data technology wherever it's used. Whether there is a database built on it or used directly, the HDFS is fundamental to big data.

This is how it works.

Large files are distributed into chunks, typically, of 64 megabytes in size and each of these chunks is stored on what is called chunk servers.

Further, chunk servers are replicated so that each chunk is stored on three different servers, typically replicated on multiple racks and in a different network subnet, to take care of all types of hardware failures. As a result of this replication, the possibility of data being lost due to hardware failure is minimized significantly.

The challenge is now to maintain consistency across all these replicas.

So, here is how that works.

When a client application, say, a map-reduce program, needs to read data, it needs to first figure out where a particular piece of data resides.

A read command typically is issued with a file name which is a, a path in the directory structure. And an offset that is how many bytes ahead in, in the file, the client wants to read.

Data about what parts of a file are on which chunk server is kept in a master node which is called the Master GFS and it's called the name node in HDFS, and the client reads the metadata from this master node. The metadata tells it where the particular offset that it is asking for resides that is which chunk server. The metadata itself is cached by the client when it first starts reading so that it doesn't continuously have to go and ask the name node or the master for information.

Further, while actually reading data, the client directly contacts the chunk server, which is listening for such requests, and serves up requests for a particular chunks of data from a file, rather than go through any centralized systems.

So, if there are many clients reading a large file, typically, they will be accessing different junk servers, and therefore, these reads are happening in parallel.

And the result input at least doesn't become a bottleneck in a parallel map reduce job. In case the reading client that is a mapper fails, to contact the chunk server, it looks up to the master data to figure out where the next replica is stored and tries again. Since there are three replicas, the chance of all of them failing is fairly low and that lends fall tolerance to the input, output process.

Of course, the tricky part comes in writing because while writing data, we have to main, make, make sure that the, each replica contains the same data all the time.

Here is how writes happen in GFS.

Of the three replicas for each chunk, one is designated as the primary replica. Could be any one, and that information is kept in the master. Of course, the master keeps pinging these replicas to make sure that they are alive. And in case, the primary is down for some reason the master node assigns a new primary, and possibly even asks for a new replica to be created for that chunk. Now, when writing data, the client application sends the data to be written to all three replicas for a particular chunk.

One of them is primary, and it figures out where to write this data, assigns an offset to write the data, such as typically the end of file, and sends this offset to the other two replicas.

If these replicas succeed in writing at that particular point, they tell the primary that they have written, and the write succeeds with the primary informing the client that the operation has successfully completed. On the other hand, if some of the replicas failed to write at the designated offset, which can happen, for example, because of a bad disc sector. Then, they return a failure to the primary, and the primary retries to write at another offset, could be beyond the end of the file, for example, and tries again until it succeeds at all replicas.

As a result of this process, large bulk writes can be done in parallel on a large file by multiple processors, typically reducers, writing the output of a map-reduce job, while still ensuring that three replicas of every chunk are maintained synchronously and with the same state. Gfs and HDFS are the foundation for all big data technology. Gfs, of course, is proprietary to Google, and is internally used. Hdfs was developed at Yahoo, and opened sourced as part of the Hadoop Distribution and has now become synonymous with map-reduce and large scale computing using big data.

So, to summarize, what the GFS distributed file system architecture is really good at is supporting multiple, parallel reads and writes from a large number of processors. The reads are arbitrary and random access but the writes are best done when they are appends or writing to the end of a large file.

Because the architecture relies on the primary replica for a chunk deciding the order, in which multiple append requests are processed. The data is always consistent, though not necessarily predictable in terms of which processor data is written first. That normally doesn't matter because the data is fairly independent, especially in map-reduced output.

At the same time, it's important to realize that random writes in the middle of some file, while can be handled using the GFS architecture, exactly the same way as we have described, are not as efficient as bulk writes towards the end of the file. Imagine a large number of reducers writing their output to a file, which will eventually become large. As soon as a reducer decides to write say 64 MB chunk, it has to figure out that this is going to be a new chunk. And inform the master or the name node, that it's writing a new chunk. Similarly, other reducers figure out that they're writing new chunks, and inform the master. The master's updates its metadata, and all the writes happen in parallel. The master does become a bit of a bottleneck, but because the writes are fairly large, that doesn't normally create a problem.

On the other hand, if we have random writes to the middle of a file, The challenge then becomes, what happens when a chunk essentially overflows and impinges on another, next, the next chunk in the file.

This can create problems in the way the file is laid out and requires much more synchronization.

Without going into too much detail, it's also shown in the paper on GFS that the degree of replica consistency that one gets with large bulk of pens is much stronger than the degree of replica consistency than we get with random writes and parallel. Nonetheless even with a reduced degree of consistencies, each reader, in the GFS architecture always sees a consistent data regardless of which replica it ends up reading from. That's one of the powers of the GFS architecture. Still, GFS and HDFS, while supporting bulk parallel reads and writes are nevertheless file system architectures and not databases as we have come to understand them over the past 30 to 40 years.

Much of the current debate in the big data and traditional BI communities is about databases. The big data databases are built on top of distributed file systems like GFS and HDFS. But before we turn to these, let's first take a look at traditional databases and see why they were developed and how they've actually been used.

**Relational database**

A relational database is a collection of data items organized as a set of formally described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables.

The standard user and application program interface to a relational database is the structured query language (SQL). SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports.

In addition to being relatively easy to create and access, a relational database has the important advantage of being easy to extend. After the original database creation, a new data category can be added without requiring that all existing applications be modified.

A relational database is a set of tables containing data fitted into predefined categories. Each table (which is sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns. For example, a typical business order entry database would include a table that described a customer with columns for name, address, phone number, and so forth. Another table would describe an order: product, customer, date, sales price, and so forth. A user of the database could obtain a view of the database that fitted the user's needs. For example, a branch office manager might like a view or report on all customers that had bought products after a certain date. A financial services manager in the same company could, from the same tables, obtain a report on accounts that needed to be paid.

When creating a relational database, you can define the domain of possible values in a data column and further constraints that may apply to that data value. For example, a domain of possible customers could allow up to ten possible customer names but be constrained in one table to allowing only three of these customer names to be specifiable.

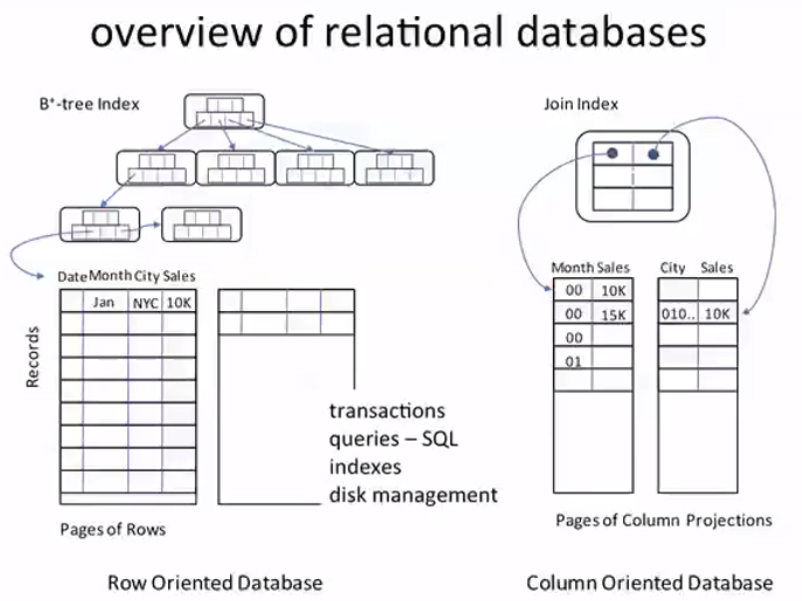
The definition of a relational database results in a table of metadata or formal descriptions of the tables, columns, domains, and constraints.

The major problem that relational databases is address is that of transaction processing, where multiple clients or users are trying to insert or update different records in the same table, and their actions have to be protected from each other so that consistency of some form is maintained.

As the result the database management systems build in many features like locking and other forms of consistency control to ensure that transactions are effectively isolated from each other. And this is the really primary function of the relational database.

The second function of course is queries. Where the user should be able to get access to any record based on a combination of where clauses or a combination of conditions that the record needs to satisfy.

Indexes, such as the b+3 index depicted here, are used to speed up access to records, so that SQL queries can actually be efficient. The b+ trees is the most common form of index, which is essentially like a binary tree, except for the fact that the interior nodes don't actually store data values. They only store key values.



For the most part when the data was reasonably small, queries using indexes could be performed fairly efficiently on exactly the same data stores where transactions would insert an update data. The row-oriented data stores were perfectly fine for both transaction processing and answering questions about data.

As, the volume of data grew and in particular the width of data in terms of the number of columns that one would store per record becomes very large. And clearly the amount of storage required grows.

But at the same time, what also happens is that a query that only requires data from, say, two columns, ends up needing to fetch data, about maybe 100 columns, just because they happen to come along with the disk block as it is fetched. This makes query processing extremely inefficient, both from a storage perspective as well as performance perspective, and for such situations, the column oriented databases have been developed in the past decade or so.

Especially to take care of situations where the number of columns is very large and only queries need to be supported rather than transactions. In a column-oriented database unlike a row store, sets of columns are stored together in disk blocks.

In other words, a particular record is actually split across in multiple disk blocks with different sets of columns occurring in different blocks.

Within each block, the data is sorted along particular sets of columns. So, that unlike a row store where data essentially gets stored as it's inserted, the column-oriented database can process the data from a row store into different sort orders for different columns.

Of course, by doing so, one sort of loses the identity of an individual record and therefore a column-oriented database needs to create extra information in the form of what is called a join index where different rows in different disk blocks are pulled together to indicate that they actually form the same record. Column stores are ideally used for what are known as analytical processing queries.

Not that such queries could not be done using rows stores as well. It's just that when you have large numbers of columns, column stores become more efficient. Now, this is an example of a OLAP or Online Analytical Processing query.

## NoSQL Database

NoSQL Database is used to refer a non-SQL or non relational database.

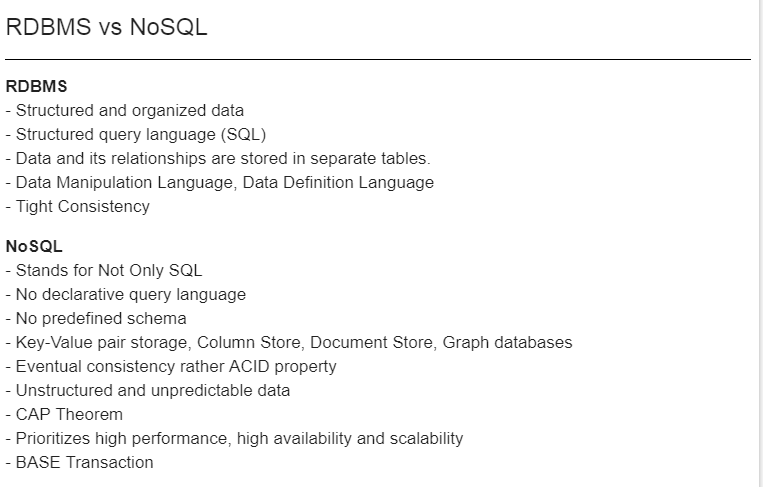
NoSQL is a non-relational database management systems, different from traditional relational database management systems in some significant ways. It is designed for distributed data stores where very large scale of data storing needs (for example Google or Facebook which collects terabits of data every day for their users). These type of data storing may not require fixed schema, avoid join operations and typically scale horizontally.

**Advantages**:

* High scalability
* Distributed Computing
* Lower cost
* Schema flexibility, semi-structure data
* No complicated Relationships

**Disadvantages**

* No standardization
* Limited query capabilities (so far)
* Eventual consistent is not intuitive to program for



One of the first no sequel databases was Google's Big Table and its, Hadoop equivalent, which is called HBase.

Big Table is, in many ways, probably the origin of the term big data. This is one of the earliest papers describing this was almost three or four years ago and it is Google's implementation on top of the distributed GFS file system, which led to the development of many no sequel databases in the first place.

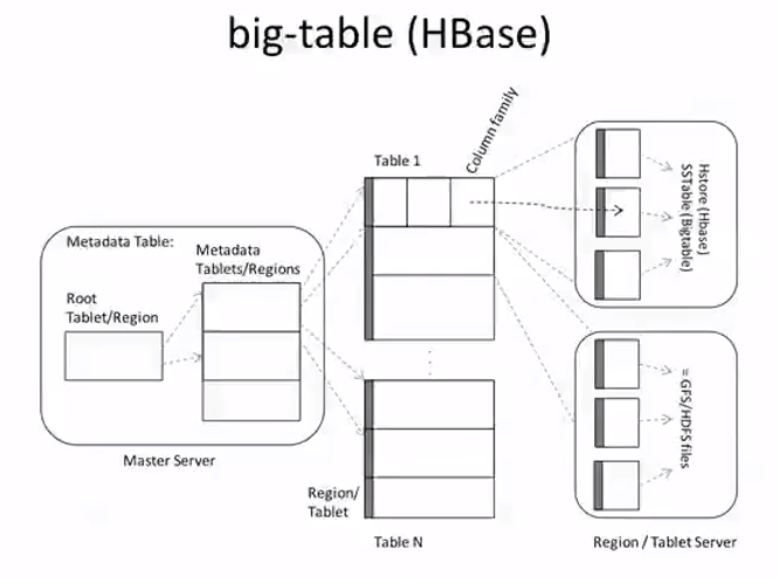
A big table, table is distributed across many different servers by row first. So that table is broken up into many tablets, each containing multiple rows. The tablets are called regions by the way in H base tablet is the GFS term. Each tablet in turn, is broken up into column families, each containing a set of columns from among those in that row. Each column family which contains.

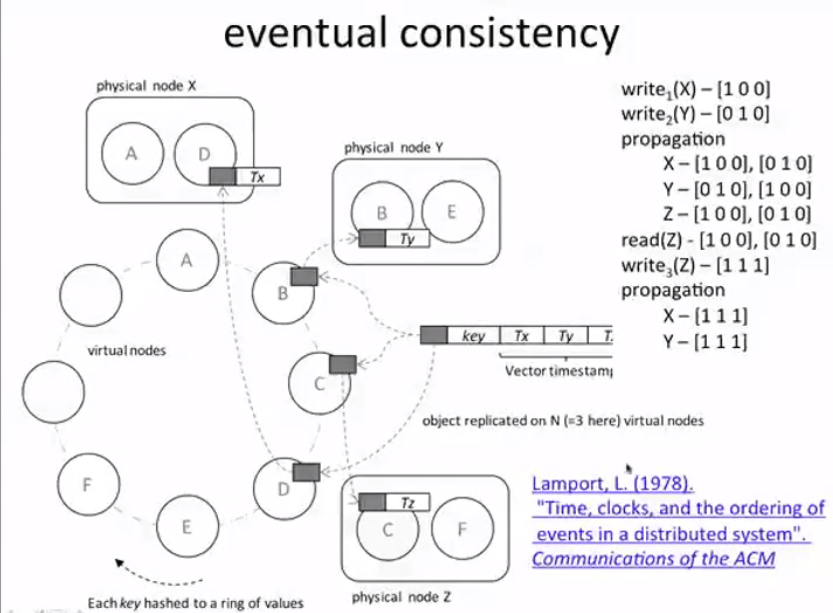
A particular set of columns spanning multiple rows, is stored in a chunk of the distributed file system the GFS or HGFS.

Each, such chunk is served by, a tablet server which takes care of three separate replicas as they are maintained in GFS as we have seen before, and the tablet servers together, form the entire big table. Of course in order to access any particular record, we need to know which tablet server that particular record and column that we are looking for falls in and for that purpose there is a metadata table which keeps track of which tablet server a particular row, lies in.

The metadata table itself is another big table and is therefore cons-, comprises of tablets or regions which are maintained on, a separate set of tablets servers.

One particular tablet is the root tablet. So all our searches start from there. So to look for a particular row, and column. One looks up the tablet, root tablet. Which will tell us which child's tablet, among the metadata tablets contain the information about that particular row column. The child tablet will in turn tell us which tablet server to look for, and there we can pick up that chunk where the required row and column family is actually stored.





Let's assume that there is a record which is mapped to three physical nodes x, y and z.

And now, let's see what happens when we try to write a record, which is replicated at three locations. Suppose we happen to write this record at x, we associate it with that at time stamp 1,0,0.

So, this is a vector time stamp consisting of three elements, which tells us that this was written at time stamp one, at physical note X, and the time stamps, the other notes are maintained at zero. Next, we may want to write it again, again at physical note x, in which case, we simply update the vector times stamp to 2,0,0 and forget about the old 1,0,0 version which was written earlier.

That's because in vector terms, 2,0,0 is greater than 1,0,0 and that at least one of the terms is greater than the one in 1,0,0 and the others are at least equal to the same in this vector. On the other hand, suppose we were to first write it at location x and the next write for some reason happened at location y just because of parallelism.

The vector time stamp for this write would then be 0,1,0 because the value it is writing had, times, times, 0,0,0, and it's writing it at y, so you, it is assigned a time stamp of 0,1,0 now.

Next, this right is propagated to all the replicas, both rights are propagated to the other two replicas, so x gets the value from y 0,1,0 and y gets the value from x and zed gets the value from both x and y. Now, in each of these cases, neither of the two values of a record are greater than each other in vector term.

So, each replica has to maintain both the values until something more interesting happens in the future. For example, now if we read the value z, where we read both these replicas 1,0,0 and 0,1,0 from the location z And now, we do some computation, and write a fresh value.

Because we read both these replicas with time stamps 1,0,0 and 0,1,0 figured out which one to use using some application logic and then wrote a value, the value written has a time stamp 1,1,1 because it's based on values that have seen 1,0,0 and 0,1,0 as time stamps. Now, 1,1,1 is greater than both of these so the value written overrides the previous two and they can be discarded.

And when this value propagates to x and y, the corresponding older values also get discarded. As a result, what we achieved is eventual consistency in that at some point of time, when you read a record you might get two values instead of just one because you are getting an inconsistent state and its up to you, as the application, to resolve that inconsistency usually using some knowledge of what these values are all about.