ANNAI WOMEN'S COLLEGE

PUNNAMCHATHRAM, KARUR - 639 136.

(Affiliated to Bharathidasan University)

DEPARTMENT OF COMPUTER SCIENCE

Class : II M.Sc Computer Science – IV Semester
Subject Title : Big Data Analytics
Subject Code : P16CSE5A
Staff Name : K.Anitha MCA.,M.Phil.,
Designation : Asst. Professor

Big Data

In the business landscape of today, data management can be a major determinant of whether you succeed or fail. Most businesses have begun to realize the importance of incorporating strategies that can transform them through the application of big data. In this endeavor, businesses are realizing that big data is not simply a single technology or technique. Rather, big data is a trend that stretches across numerous fields in business and technology.

Big Data is the term used to refer to initiatives and technologies that comprise of data that is too diverse, fast evolving, and vast for ordinary technologies, infra- structure, and skills to address exhaustively. That is; the volume, velocity and variety of the data is far too great. Despite the complexity of this data, advances in technology are allowing businesses to draw value from big data.

For example, in your businesses can be positioned to track consumer web clicks in order to identify consumers' behavioral trends and modify the business's campaigns, advertisements, and pricing to fit the consumers' persona.

An additional example would be where energy service providers assess household consumption levels in order to predict impending outages and promote more efficient energy consumption.

Additionally, health provision bodies may be able to monitor the spread as well as the emergence of illnesses by analyzing social media data. There are numerous applications of big data, the most noteworthy of which will be discussed a little later in the article.

Big Data involves the creation of large amounts of complex data, its storage, its retrieval, and finally its analysis.

Characteristics of Big Data

The following are the **three Vs of big data**.

- **Volume.** Two decades ago, typical computers may have had about ten gigabytes of memory. Today, however, social media platforms such as Facebook will take in over half a billion terabytes of data on a daily basis. Similarly, Boeing airplanes generate hundreds of terabytes in flight data in a single flight. The wide spread use of smartphones and tablets results in the generation of billions of terabytes of consistently updated data feeds that are of infinitely diverse genres.
- **Velocity**. Clickstreams capture user behavior at millions of events each second. For example, stock trading market changes are reflected within microseconds. Computer processes exchange data between billions of gadgets, infrastructure, and sensors in order to generate accurate and applicable data in real-time. For example, on-line gaming systems support millions of users operating concurrently and with each producing multiple inputs every second.
- **Variety**. Big data does not just refer to numbers and dates, big data is all that inclusive of audio, video, unstructured text, social media information, and so much more. Database systems of about two decades ago had been designed to address a smaller volume of structured data, slower, and fewer updates. They were designed to process structured and predictable forms of data. These traditional databases were also designed to operate on single servers, which would make an increase in capacity an expensive endeavor. Programs and applications have evolved to serve large volumes of users and the use of the olden databases has become a liability for most businesses as opposed to an asset. Big Data databases, for example MongoDB, solve these issues and avail businesses great value.

Map Reduce

What is MapReduce?

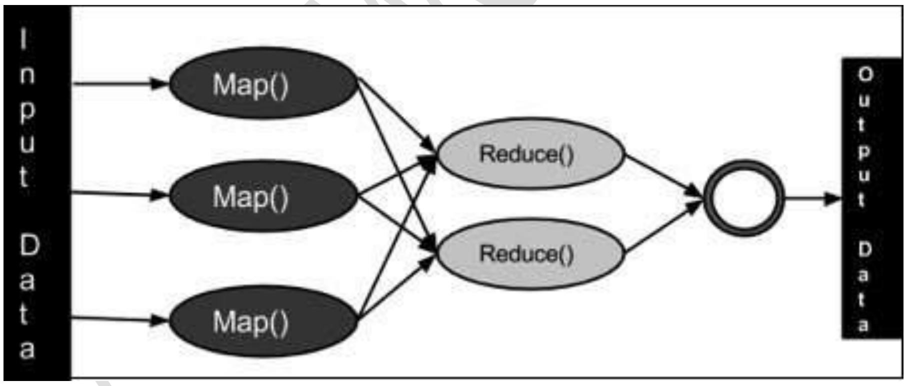
MapReduce is a framework using which we can write applications to process huge amounts of data, in parallel, on large clusters of commodity hardware in a reliable manner.

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce. Map takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs). Secondly, reduce task, which takes the output from a map as an input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

The major advantage of MapReduce is that it is easy to scale data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called mappers and reducers. Decomposing a data processing application into *mappers* and *reducers* is sometimes nontrivial. But, once we write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to use the MapReduce model.

The Algorithm

- Generally MapReduce paradigm is based on sending the computer to where the data resides!
- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
    - **Map stage** − The map or mapper's job is to process the input data. Generally the input data is in the form of file or directory and is stored in the Hadoop file system (HDFS). The input file is passed to the mapper function line by line. The mapper processes the data and creates several small chunks of data.
    - **Reduce stage** − This stage is the combination of the **Shuffle**stage and the **Reduce** stage. The Reducer's job is to process the data that comes from the mapper. After processing, it produces a new set of output, which will be stored in the HDFS.
- During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
- The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
- Most of the computing takes place on nodes with data on local disks that reduces the network traffic.
- After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.



Inputs and Outputs (Java Perspective)

The MapReduce framework operates on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types.

The key and the value classes should be in serialized manner by the framework and hence, need to implement the Writable interface. Additionally, the key classes have to implement the

Writable-Comparable interface to facilitate sorting by the framework. Input and Output types of a **MapReduce job** − (Input) <k1, v1> → map → <k2, v2> → reduce → <k3, v3>(Output).

Input	**Output**	
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

Terminology

- **PayLoad** − Applications implement the Map and the Reduce functions, and form the core of the job.
- **Mapper** − Mapper maps the input key/value pairs to a set of intermediate key/value pair.
- **NamedNode** − Node that manages the Hadoop Distributed File System (HDFS).
- **DataNode** − Node where data is presented in advance before any processing takes place.
- **MasterNode** − Node where JobTracker runs and which accepts job requests from clients.
- **SlaveNode** − Node where Map and Reduce program runs.
- **JobTracker** − Schedules jobs and tracks the assign jobs to Task tracker.
- **Task Tracker** − Tracks the task and reports status to JobTracker.
- **Job** − A program is an execution of a Mapper and Reducer across a dataset.
- **Task** − An execution of a Mapper or a Reducer on a slice of data.
- **Task Attempt** − A particular instance of an attempt to execute a task on a SlaveNode.

What is MapReduce in Hadoop?

MapReduce is a programming model suitable for processing of huge data. Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++. MapReduce programs are parallel in nature, thus are very useful for performing large-scale data analysis using multiple machines in the cluster.

MapReduce programs work in two phases:

1. Map phase
2. Reduce phase.

An input to each phase is **key-value** pairs. In addition, every programmer needs to specify two functions: **map function** and **reduce function**.

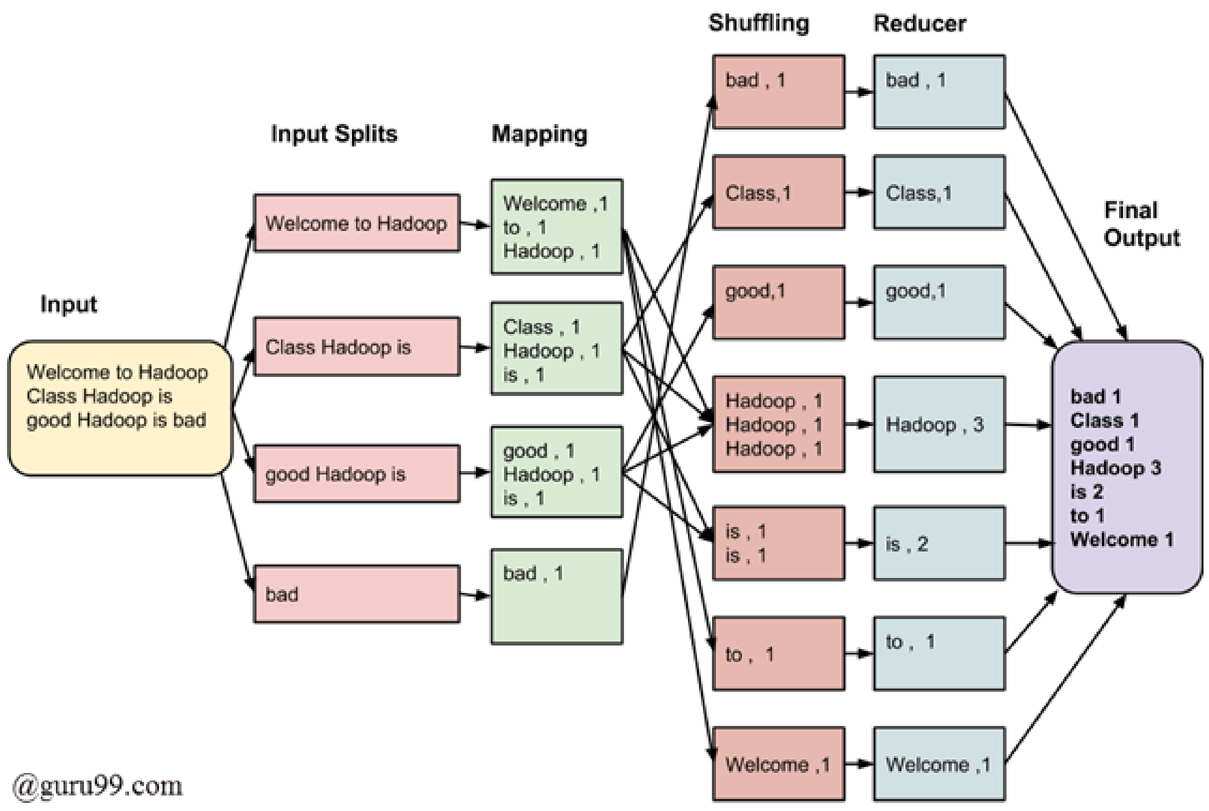
How MapReduce Works? Complete Process

The whole process goes through four phases of execution namely, splitting, mapping, shuffling, and reducing.

Let's understand this with an example –

Consider you have following input data for your Map Reduce Program

```
Welcome to Hadoop Class
Hadoop is good
Hadoop is bad
```



The final output of the MapReduce task is

bad	1
Class	1
good	1

Hadoop	3
is	2
to	1
Welcome	1

The data goes through the following phases

Input Splits:

An input to a MapReduce job is divided into fixed-size pieces called **input splits** Input split is a chunk of the input that is consumed by a single map

Mapping

This is the very first phase in the execution of map-reduce program. In this phase data in each split is passed to a mapping function to produce output values. In our example, a job of mapping phase is to count a number of occurrences of each word from input splits (more details about input-split is given below) and prepare a list in the form of <word, frequency>

Shuffling

This phase consumes the output of Mapping phase. Its task is to consolidate the relevant records from Mapping phase output. In our example, the same words are clubed together along with their respective frequency.

Reducing

In this phase, output values from the Shuffling phase are aggregated. This phase combines values from Shuffling phase and returns a single output value. In short, this phase summarizes the complete dataset.

In our example, this phase aggregates the values from Shuffling phase i.e., calculates total occurrences of each word.

MapReduce Architecture explained in detail

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits because the time taken to process a split is small as compared to the time taken for processing of the whole input. When the splits are smaller, the processing is better to load balanced since we are processing the splits in parallel.

- However, it is also not desirable to have splits too small in size. When splits are too small, the overload of managing the splits and map task creation begins to dominate the total job execution time.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 64 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is, to avoid replication which takes place in case of HDFS store operation.
- Map output is intermediate output which is processed by reduce tasks to produce the final output.
- Once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication becomes overkill.
- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task. Map output is transferred to the machine where reduce task is running.
- On this machine, the output is merged and then passed to the user-defined reduce function.
- Unlike the map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes). So, writing the reduce output

How MapReduce Organizes Work?

Hadoop divides the job into tasks. There are two types of tasks:

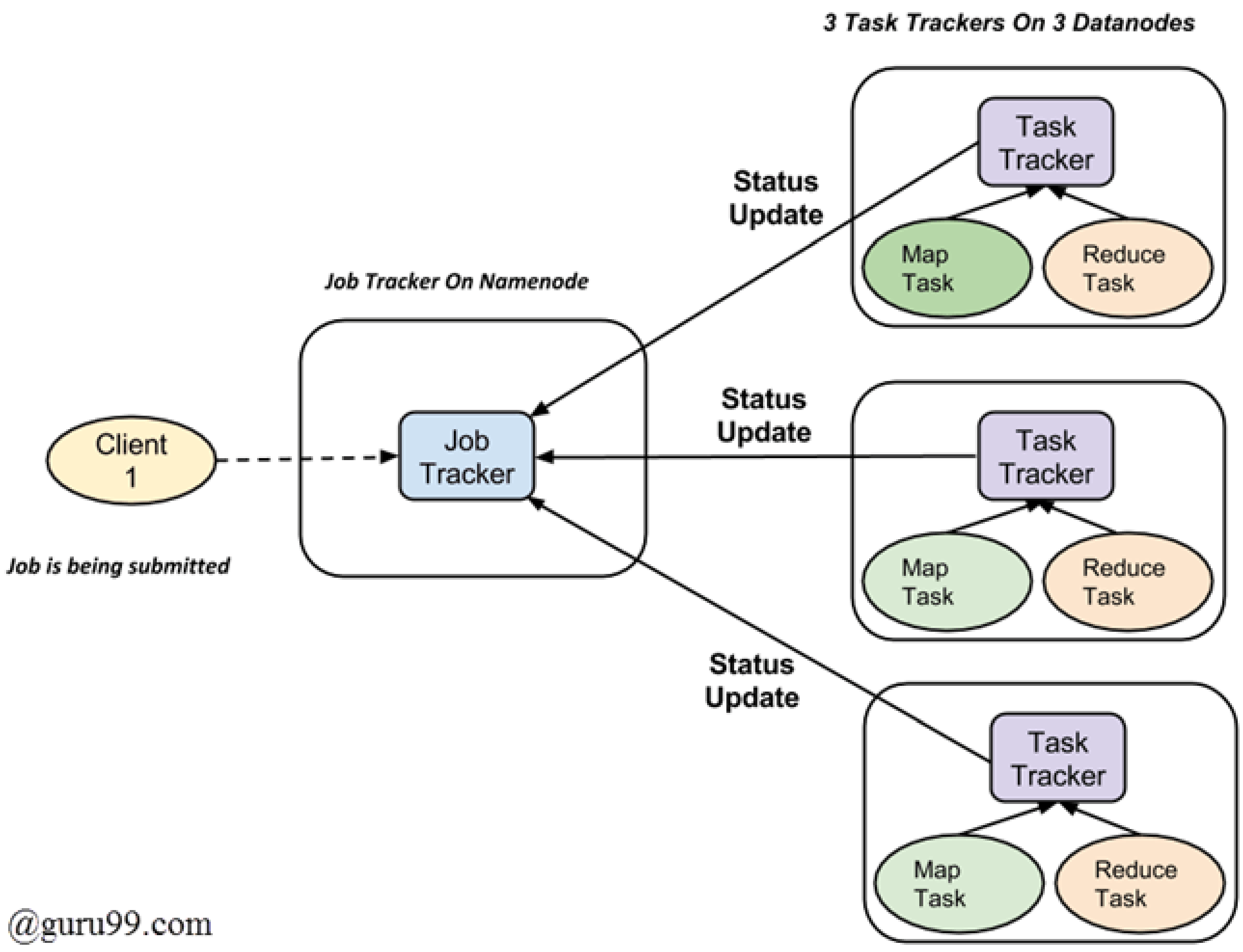
1. **Map tasks** (Splits & Mapping)
2. **Reduce tasks** (Shuffling, Reducing)

as mentioned above.

The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. **Jobtracker**: Acts like a **master** (responsible for complete execution of submitted job)
2. **Multiple Task Trackers**: Acts like **slaves,** each of them performing the job

For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.



- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- In addition, task tracker periodically sends **'heartbeat'** signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job. In the event of task failure, the job tracker can reschedule it on a different task tracker.

MONGO DB

MongoDB is a document-oriented NoSQL database used for high volume data storage. MongoDB is a database which came into light around the mid-2000s. It falls under the category of a NoSQL database.

MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era. No database makes you more productive.

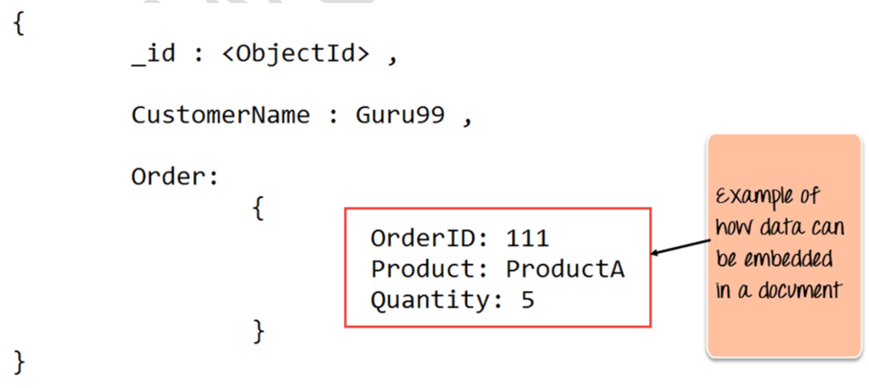
MongoDB Features

1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. As seen in the introduction with NoSQL databases, the rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
    1. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

MongoDB Example

The below example shows how a document can be modeled in MongoDB.

1. The _id field is added by MongoDB to uniquely identify the document in the collection.
2. What you can note is that the Order Data (OrderID, Product, and Quantity) which in RDBMS will normally be stored in a separate table, while in MongoDB it is actually stored as an embedded document in the collection itself. This is one of the key differences in how data is modeled in MongoDB.



```
{
        _id : <ObjectId> ,

        CustomerName : Guru99 ,

        Order:
                {
                        OrderID: 111
                        Product: ProductA
                        Quantity: 5
                }
}
```

Key Components of MongoDB Architecture

Below are a few of the common terms used in MongoDB

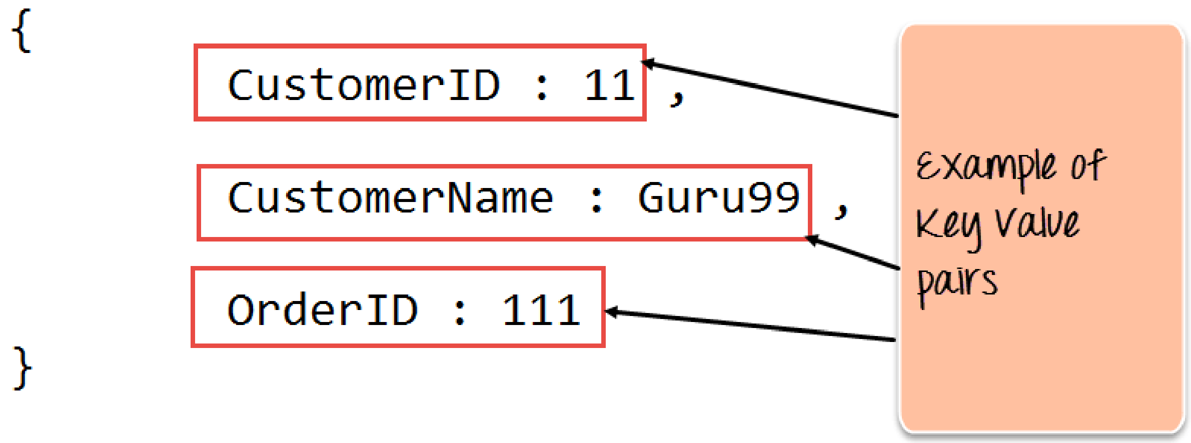
1. **_id** – This is a field required in every MongoDB document. The _id field represents a unique value in the MongoDB document. The _id field is like the document's primary key. If you create a new document without an _id field, MongoDB will automatically create the field. So for example, if we see the example of the above customer table, Mongo DB will add a 24 digit unique identifier to each document in the collection.

2. **Collection** – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL.

_Id	**CustomerID**	**CustomerName**	**OrderID**
563479cc8a8a4246bd27d784	11	Guru99	111
563479cc7a8a4246bd47d784	22	Trevor Smith	222
563479cc9a8a4246bd57d784	33	Nicole	333

   A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.
3. **Cursor** – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
4. **Database** – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.
5. **Document** - A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.
6. **Field** - A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases.

   The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair's defined in the document.



7. **JSON** – This is known as JavaScript Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

Just a quick note on the key difference between the _id field and a normal collection field. The _id field is used to uniquely identify the documents in a collection and is automatically added by MongoDB when the collection is created.

Features of MongoDB

1. Document-oriented – Since MongoDB is a NoSQL type database, instead of having data in a relational type format, it stores the data in documents. This makes MongoDB very flexible and adaptable to real business world situation and requirements.
2. Ad hoc queries - MongoDB supports searching by field, range queries, and regular expression searches. Queries can be made to return specific fields within documents.
3. Indexing - Indexes can be created to improve the performance of searches within MongoDB. Any field in a MongoDB document can be indexed.
4. Replication - MongoDB can provide high availability with replica sets. A replica set consists of two or more mongo DB instances. Each replica set member may act in the role of the primary or secondary replica at any time. The primary replica is the main server which interacts with the client and performs all the read/write operations. The Secondary replicas maintain a copy of the data of the primary using built-in replication. When a primary replica fails, the replica set automatically switches over to the secondary and then it becomes the primary server.
5. Load balancing - MongoDB uses the concept of sharding to scale horizontally by splitting data across multiple MongoDB instances. MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

Data Modelling in MongoDB

As we have seen from the Introduction section, the data in MongoDB has a flexible schema. Unlike in SQL databases, where you must have a table's schema declared before inserting data, MongoDB's collections do not enforce document structure. This sort of flexibility is what makes MongoDB so powerful.

When modeling data in Mongo, keep the following things in mind

1. What are the needs of the application – Look at the business needs of the application and see what data and the type of data needed for the application. Based on this, ensure that the structure of the document is decided accordingly.
2. What are data retrieval patterns – If you foresee a heavy query usage then consider the use of indexes in your data model to improve the efficiency of queries.
3. Are frequent inserts, updates and removals happening in the database? Reconsider the use of indexes or incorporate sharding if required in your data modeling design to improve the efficiency of your overall MongoDB environment.

Difference between MongoDB & RDBMS
Below are some of the key term differences between MongoDB and RDBMS

RDBMS	MongoDB	Difference
Table	Collection	In RDBMS, the table contains the columns and rows which are used to store the data whereas, in MongoDB, this same structure is known as a collection. The collection contains documents which in turn contains Fields, which in turn are key-value pairs.
Row	Document	In RDBMS, the row represents a single, implicitly structured data item in a table. In MongoDB, the data is stored in documents.
Column	Field	In RDBMS, the column denotes a set of data values. These in MongoDB are known as Fields.
Joins	Embedded documents	In RDBMS, data is sometimes spread across various tables and in order to show a complete view of all data, a join is sometimes formed across tables to get the data. In MongoDB, the data is normally stored in a single collection, but separated by using Embedded documents. So there is no concept of joins in MongoDB.

Difference b/w Mongo and RDBMS

1. Relational databases are known for enforcing data integrity. This is not an explicit requirement in MongoDB.
2. RDBMS requires that data be normalized first so that it can prevent orphan records and duplicates Normalizing data then has the requirement of more tables, which will then result in more table joins, thus requiring more keys and indexes.

   As databases start to grow, performance can start becoming an issue. Again this is not an explicit requirement in MongoDB. MongoDB is flexible and does not need the data to be normalized first.

Big Data Serialization

Serialization is the process of converting structured data into its raw form. **Deserialization** is the reverse process of reconstructing structured forms from the data's raw bit stream form.

In Hadoop, different components talk to each other via **Remote Procedure Calls** (**RPCs**). A caller process serializes the desired function name and its arguments as a byte stream before sending it to the called process. The called process deserializes this byte stream, interprets the function type, and executes it using the arguments that were supplied. The results are serialized and sent back to the caller. This workflow naturally calls for fast serialization and deserialization. Network bandwidth is at a premium and requires the serialized representation of the function name and its arguments to have the smallest possible payload. Different processes might evolve differently, and the entire...

A container is nothing but a data structure, or an object to store data in. When we transfer data over a network the container is converted into a byte stream. This process is referred to as serialization. The reverse, that is converting a byte stream into a container, is called deserialization.

Serialization Formats in Hadoop

The information shown in the table below has been collected from various sources, notably Hadoop Application Architectures, Hadoop – The Definitive Guide, Apache Hive Essentials, and the documentation of the respective contestants:

- Writables
- Thrift
- Protocol buffers
- Avro
- RCFile API
- ORC
- Parquet

3.3. Big data serialization formats

Unstructured text works well when you’re working with scalar or tabular data. Semistruct
ured text formats such as XML and JSON can model more sophisticated data structures that i
nclude composite fields or hierarchical data. But when you’re working with big data volumes,
you’ll need serialization formats with compact serialized forms that natively support partitioni
ng and have schema evolution features.

In this section we’ll compare the serialization formats that work best with big data in Map
Reduce and follow up with how you can use them with MapReduce.

3.3.1. Comparing SequenceFile, Protocol Buffers, Thrift, and Avro

In my experience, the following characteristics are important when selecting a data seriali
zation format:

· ***Code generation*** —Some serialization formats are accompanied by libraries with co
de-generation abilities that allow you to generate rich objects, making it easier for you to inter
act with your data. The generated code also provides the added benefit of type-safety to make
sure that your consumers and producers are working with the right data types.

· ***Schema evolution*** —Data models evolve over time, and it’s important that your dat
a formats support your need to modify your data models. Schema evolution allows you to add
, modify, and in some cases delete attributes, while at the same time providing backward and f
orward compatibility for readers and writers.

· ***Language support*** —It’s likely that you’ll need to access your data in more than on e programming language, and it’s important that the mainstream languages have support for a data format.

· ***Transparent compression*** —Data compression is important given the volumes of da ta you’ll work with, and a desirable data format has the ability to internally compress and dec ompress data on writes and reads. It’s a much bigger headache for you as a programmer if the data format doesn’t support compression, because it means that you’ll have to manage compr ession and decompression as part of your data pipeline (as is the case when you’re working w ith text-based file formats).

· ***Splittability*** —Newer data formats understand the importance of supporting multiple parallel readers that are reading and processing different chunks of a large file. It’s crucial tha t file formats contain synchronization markers (and thereby support the ability for a reader to perform a random seek and scan to the start of the next record).

· ***Support in MapReduce and the Hadoop ecosystem*** —A data format that you select must have support in MapReduce and other critical Hadoop ecosystem projects, such as Hive. Without this support, you’ll be responsible for writing the code to make a file format work wi th these systems.

Let’s look at each of these formats in more detail.

SequenceFile

The SequenceFile format was created to work with MapReduce, Pig, and Hive, and theref ore integrates well with all of those tools. Its shortcomings are mainly its lack of code generat ion and versioning support, as well as limited language support.

Protocol Buffers

The Protocol Buffers format has been used heavily by Google for interoperability. Its stre ngths are its versioning support and compact binary format. Downsides include its lack of sup port in MapReduce (or in any third-party software) for reading files generated by Protocol Bu ffers serialization. Not all is lost, however; we’ll look at how Elephant Bird uses Protocol Buf fers serialization within a higher-level container file in section 3.3.3.

Thrift

Thrift was developed at Facebook as a data-serialization and RPC framework. It doesn’t h ave support in MapReduce for its native data-serialization format, but it can support different wire-level data representations, including JSON and various binary encodings. Thrift also incl udes an RPC layer with various types of servers, including a nonblocking implementation. W e’ll ignore the RPC capabilities for this chapter and focus on the data serialization.

Avro

The Avro format is Doug Cutting’s creation to help address the shortcomings of Sequence File.

Parquet

Parquet is a columnar file format with rich Hadoop system support, and it works well with data models such as Avro, Protocol Buffers, and Thrift. Parquet is covered in depth in section 3.4.

Based on certain evaluation criteria, Avro seems to be the best fit as a data serialization fr
amework in Hadoop. SequenceFile is a close second due to its inherent compatibility with Ha
doop (it was designed for use with Hadoop).

You can review a useful jvm-serializers project at https://github.com/eishay/jvm-serializer
s/wiki/, which runs various benchmarks to compare file formats based on items such as seriali
zation and deserialization times. It contains benchmarks for Avro, Protocol Buffers, and Thrif
t, along with a number of other frameworks.

After looking at how the various data-serialization frameworks compare, we'll dedicate th
e next few sections to working with them. We'll start off with a look at SequenceFile.

First, let's define **what serialization is ?** In Java, serialization is linked to *java.io.Serializable* interface and possibility to convert and reconvert object to byte stream. But regarding to Big Data systems where data can come from different sources, written in different languages, this solution has some drawbacks, as a lack of portability or maintenance difficulty.
In Big Data, serialization also refers to converting data into portable structure as byte streams. But it has another goal which is schema control. Thanks to schema describing data structure, data can be validated on writing phase. It avoids to have some surprises when data is read and, for example, a mandatory field is missing or has bad type (int instead of array).

Additionally, serialization helps to execute Big Data tasks efficiently. Unlike popular formats as JSON or XML, serialized data is splittable easier. And splittability can be used for example by MapReduce to process input data divided to input splits.

Previously mentioned schema control involves another advantage of serialization frameworks - versioning. Let's imagine that one day our object has 5 fields and one week later, it has already 10 fields. To handle the change we could, for example, map new fields with default values or use old schema file for data deserialization.

Also, in some cases, serialized data takes less place that standard JSON or XML files.

To resume, we can list following points to characterize serialization in Big Data systems:

- splittability - easier to achieve splits on byte streams rather than JSON or XML files
- portability - schema can be consumed by different languages
- versioning - flexibity to define fields with default values or continue to use old schema version
- data integrity - serialization schemas enforces data corecteness. Thanks to them, errors can be detected earlier, when data is written.

Avro as use case

There are several main serialization frameworks available, among others: Avro, Thrift and Protocol Buffers. After analyzing some of implemented features, I decided to play a little with Apache Avro.

This choice was dictated by use flexibility (optional code generation), language support, splittability and the fact that in the most of Big Data books I studied, only Apache Thrift was widely presented. But Apache Avro will be the subject of another post.

In this article we can learn some points about serialization use in Big Data systems. We can see that it helps to keep data consistent, but also improves data processing with splittability and compression features.