

**Big Data MONGODB Simplified**



**Big Data, NOSQL  
Architecting MONGODB**

**By**

Navin Sabharwal

Shakuntala Gupta Edward

# Big Data, NoSQL

## Architecting MongoDB

*Dedicated to the people I love and the God I trust.*

— Navin Sabharwal

*Dedicated to people who made my life worth living  
and carved me into an individual I am today and to  
God who shades every step of my life.*

— Shakuntala Gupta Edward

# Contents at a Glance

About the Authors

Acknowledgments

# Preface

## About this book

- 1: Big Data
- 2: NOSQL
- 3: Introducing MongoDB
- 4: MongoDB Data Model
- 5: MongoDB – Installation and Configuration
- 6: Using Mongo Shell
- 7: MongoDB Explained
- 8: Administering MongoDB
- 9: MongoDB Use Cases
- 10: MongoDB How To's

# Contents

## [Big Data](#)

[Introduction](#)

[BIG DATA](#)

[Facts on Big Data](#)

[Big Data Sources](#)

[Three V's of Big Data](#)

[Usage of big data](#)

[BIG DATA Challenges](#)

[Legacy Systems and Big Data](#)

[Big Data Technologies](#)

[Summary](#)

## [NOSQL](#)

[Introduction](#)

[SQL](#)

[NOSQL](#)

[Definition](#)

[Brief history of NoSQL](#)

[ACID vs. BASE](#)

[CAP Theorem \(Brewer's Theorem\)](#)

[The BASE](#)

[NoSQL Advantages and Disadvantages](#)

[Advantages of NoSQL](#)

[Challenges of NoSQL](#)

[SQL vs. NoSQL Databases](#)

[Categories of NoSQL database](#)

[Summary](#)

## [Introducing MongoDB](#)

[Introduction](#)

[History](#)

[Definition](#)

[MongoDB Design Philosophy](#)

[Speed, Scalability and Agility](#)

[Non-Relational Approach](#)

[Transactional Support](#)

[JSON based Document Store](#)

[Performance vs. Features](#)

[Running the Database Anywhere](#)

[SQL Comparison](#)

[Summary](#)

## [MongoDB – Data Model](#)

[The Data Model](#)

[JSON AND BSON](#)

[The Identifier \(\\_id\)](#)

[Capped Collection](#)

[Polymorphic Schemas](#)

[Object-Oriented Programming](#)

[Schema Evolution](#)

[Summary](#)

## [MongoDB - Installation & Configuration](#)

[Select Your Version](#)

[Installing MongoDB under Linux](#)

[Installing using Repositories](#)

[Installing Manually](#)

[Installing MongoDB on Windows](#)

[Start Running MongoDB](#)

[Preconditions](#)

[Start the Service](#)

[Verifying the Installation](#)

[Using the MongoDB Shell](#)

[Securing the Deployment](#)

[Using Authentication and Authorization](#)

[Controlling access over network](#)

[Summary](#)

## [Using Mongo Shell](#)

[Part I: Basic Querying](#)

[Create and Insert](#)

[Explicit Create Collection](#)

[Insert documents using Loop](#)

[Insert with explicitly specifying \\_id](#)

[Update](#)

[Delete](#)

[Read](#)

[Using Indexes](#)

[Part II Stepping Beyond the Basics](#)

[Using Conditional Operators](#)

[Regular Expressions](#)

[MapReduce](#)

[aggregate\(\)](#)

[Part III](#)

[Relational Data Modeling and Normalization](#)

[MongoDB Document Data Model Approach](#)

[Summary](#)

## [MongoDB Explained](#)

[Architecture](#)

[Core processes](#)

[MongoDB Tools](#)

[Standalone deployment](#)

[Replication](#)

[Sharding](#)

[Production Cluster Architecture](#)

[Data Storage Model](#)

[Data File](#)

[Read and writes](#)

[How data is written Using Journaling](#)

[GridFS – The MongoDB File System](#)

[The rationale of GridFS](#)

[GridFS under the hood](#)

[Using GridFS](#)

[Indexing](#)

[Type of Indexes](#)

[Behaviors are Limitations](#)

[Summary](#)

## [Administering MongoDB](#)

[Administration Tools](#)

[mongo](#)

[Third Party Administration Tool](#)

[Backup and Recovery](#)

[Data File Backup](#)

[mongodump and mongorestore](#)

[fsync and Lock](#)

[Slave Backups](#)

[Importing and Exporting](#)

[mongoimport](#)

[mongoexport](#)

[Managing the Server](#)

[Start a Server](#)

[Stop a Server](#)

[View Log Files](#)

[Server Status](#)

[Identify and Repair Server](#)

[Identify and Repair collection level data](#)

[Monitoring MongoDB](#)

[mongostat](#)

[mongod web interface](#)

[Third-Party Plug-Ins](#)

[Summary](#)

## [MongoDB Use Cases](#)

[Use Cases](#)

[Use Case 1 - Performance Monitoring](#)

[Use Case 2 – Social Networking](#)

[Limitations and Not Applicable Ranges](#)

[Limitations](#)

[MongoDB Not applicable Range](#)

[Summary](#)

## [MongoDB How To's](#)

[How To's](#)

[Deployment](#)

[Coding](#)

[Application Response Time optimization](#)

[Data Safety.](#)

[Administration](#)

[Replication Lag](#)

[Sharding](#)

[Monitoring](#)

[Summary](#)



ॐ शान्तिः शान्तिः शान्तिः । ।

ॐ असतो मा सद्गमय  
तमसो मा ज्योतिर्गमय  
मृत्योर्मा अमृतं गमय । ।

Om Asato ma sadgamaya  
Tamaso ma jyotirgamaya  
Mrityorma amritam gamaya  
बहद रण्यक उपनिषद 1.3.28

From untruth, lead me to the truth;  
From darkness, lead me to the light;  
From death, lead me to immortality.

# About the Authors

**Navin Sabharwal** is an innovator, thought leader, author, and consultant in the areas Reporting and Analytics, RDBMS Technologies including SQL Server, Oracle, MySQL Big Data Technologies, Hadoop, MongoDB and SAP HANA. Navin has been using big data technologies in creating products and services in the areas of IT service management, product development, cloud computing, cloud lifecycle management, and social network product development.

Navin has created niche award-winning products and solutions and has filed numerous patents in diverse fields such as IT services, assessment engines, ranking algorithms, capacity planning engines, and knowledge management.

Navin holds a Masters in Information Technology and is a Certified Project Management Professional.

Navin has authored the following books: Cloud Computing First Steps (Publisher: CreateSpace, ISBN#: 978-1478130086), Apache Cloudstack Cloud Computing (Publisher: Packt Publishing, ISBN#: 978-1782160106), Cloud Capacity Management (Publisher Apress, ISBN #: 978-1430249238)

**Shakuntala Gupta Edward** has been working with database technologies since 10 years. Her experience ranges from SQL Server, Oracle Databases, Analytics platforms and Big Data technologies like MongoDB, Cassandra and SAP HANA.

Shakuntala is an accomplished architect with experience in leveraging diverse database technologies to create products and solutions in various business domains.

Shakuntala has been involved in developing products and solutions leveraging big data technologies MongoDB and Cassandra.

Shakuntala holds a Master's Degree in Computer Applications.  
The authors can be reached at [architectbigdata@gmail.com](mailto:architectbigdata@gmail.com).

# Acknowledgments

Special thanks go out to the people who have helped in creation of this book Rajeev Pratap Singh for helping with the code snippets and Dheeraj Raghav and Rajendra Prasad for their creative inputs in the design of this book.

A ton of thanks go out to Stuti Awasthi for being the initiator and inspiration for this book

The authors will like to acknowledge the creators of big data technologies and the open source community for providing such powerful tools and technologies to code with and enable products and solutions which solve real business problems easily and quickly.

## Preface

Data warehousing as an industry has been around for quite a number of years now. Relational databases have been used to store data for decades while SQL has been the de-facto language to interact with RDBMS. With the emergence of Social Networking, Internet of Things and huge volumes of unstructured data on the internet the needs of data storage, processing and analytics just exploded. Traditional RDBMS systems and storage technologies were not designed to handle such vast variety and volumes of data.

Thus was born the Big Data technologies which now power various internet scale companies and their huge amounts of data. Companies like Facebook, Twitter, Google and yahoo are leveraging the big data technologies to provide products and services at internet scale which support millions of users.

This book will help our readers to appreciate the big data technologies, their emergence, need and then we will provide a deep dive technical perspective on architecting solutions using MongoDB. The book will enable our readers to understand the key use cases where big data technologies fit in and also provide them pointers on where Big Data technologies should be used carefully or combined with traditional RDBMS technologies to provide a feasible solution.

Along with the architecture the book aims to provide a step by step guide on learning MongoDB and creating applications and solutions using MongoDB.

We sincerely hope our readers will enjoy reading the book as much as we have enjoyed writing it.

# About this book

## This book

- Acts as a guide that helps the reader in grasping the various buzz words in Big Data technologies and getting a grip over various aspects of Big Data.
- Acts as a guide for people in order to understand about NoSQL and Document based database and how they are different from the traditional relational database.
- Provides insight into architecting solutions using MongoDB, it also provides information on the limitations of MongoDB as a tool.
- Methodically covers architecture, development, administration and data model of MongoDB.
- Cites examples in order to make the users comfortable in getting started with the technology.

# What you need for this book

MongoDB supports the most popular platforms.

Download the latest stable production release of MongoDB from the [MongoDB downloads page](http://www.mongodb.org/downloads/) (<http://www.mongodb.org/downloads/>)

In this book we have focused on using MongoDB on a 64-bit Windows platform and at places have cited references on how to work with MongoDB running on Linux.

At the time of writing, the latest stable MongoDB production release is 2.4.9 (see the below Figure).



We will be using 64-bit Windows 2008 R2 and LINUX for examples of the installation process.

## **Conventions Used In the Book**

**Italic** indicates important points, commands.

*This is used to denote the Code Commands*

*This is the Output of the command.....*

*This is used for Example commands*



This icon indicates statistics figures



This icon indicates examples



This icon indicates points to be noted.



This icon indicates further reading links or references.

# Who this book is for

This book would be of interest to Programmers, Big Data Architects, Application Architects, Technology Enthusiasts, Students, Solution Experts and those wishing to choose the right big data products for their needs.

The book covers aspects on Big Data, NOSQL and details on architecture and development on MongoDB. Thus it serves the use cases of developers, architects and operations teams who work on MongoDB.

# Big Data

*"Big Data is used to describe data which has massive volume, varied structure and is generated at high velocity. This is posing challenges to the traditional RDBMS systems which is used for storing and processing data and paving way for newer approaches of processing and storing data"*

In this chapter we will talk upon the basics of Big Data, sources of Big Data and Challenges of Big Data.

We will introduce the readers to the three V's of Big Data and the limitations of traditional technologies in handling Big Data.

## Introduction

Big Data along with Cloud, Social, Analytics and Mobility is the buzz word today in the Information Technology World. Data is getting generated from varied sources in varied formats such as Video, text, speech, log files, images etc. at a very high speed.

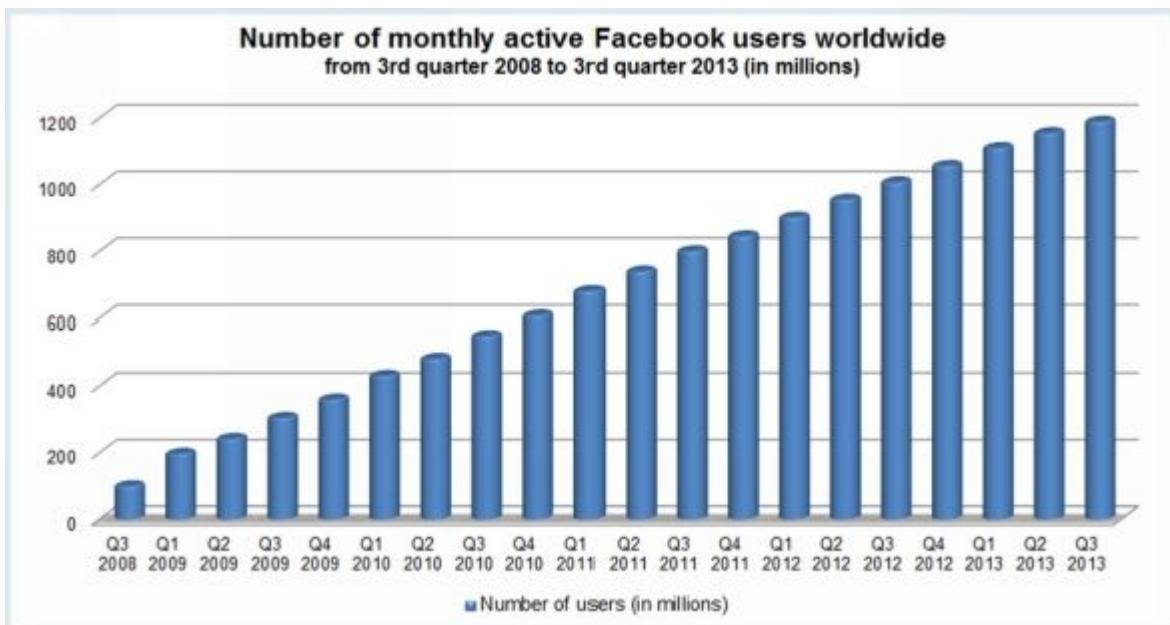
The advent of Social media sites, smartphones, and other data generating consumer devices including PCs, Laptops and Tablets is leading to an explosion of data.

The availability of Internet and the electronic devices with the masses is increasing every day and with this is increasing the number of connected devices and thus the data generated is increasing at an enormous speed.

The availability of photo and video technologies and their ease of usage are generating huge amounts of graphic data. Each second of high-definition video, for example, generates more than 2,000 times as many bytes as compared to store a single page of text. Enterprises are realizing the impact of Big Data in their business applications and environments. The potential of Big Data is enormous - from understanding the behavior of consumers to fraud detection to military applications big data is playing an ever increasing role.

The following figures depict the statistics of the social networking sites Facebook and twitter.

**Fig 1-1: Facebook Active Users Stats**



**Source: Facebook**

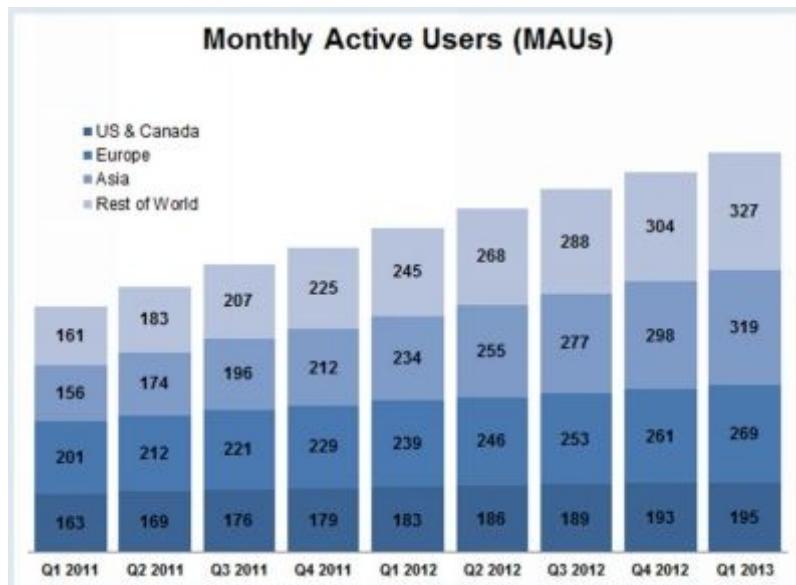


Few Stats for the year 2013:

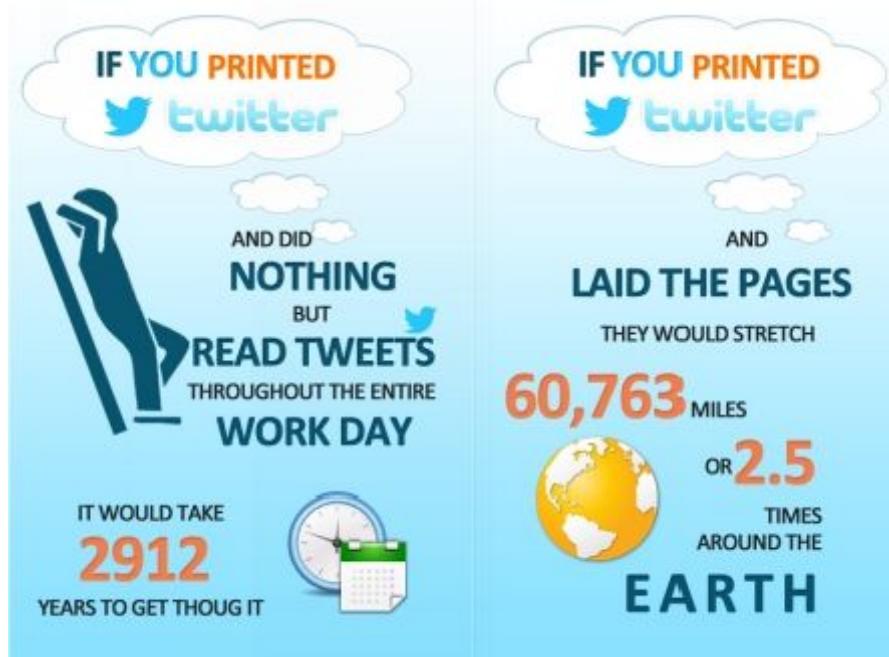
1. Worldwide, there are over 1.19 billion active Facebook users. (Source: Facebook)
2. 4.5 billion likes generated daily as of May 2013 which is a 67 percent increase from August 2012 (Source: Facebook)

3. 728 million people log onto Facebook daily, which represents a 25% increase from 2012 (Source: Facebook as of 9/2013)

**Fig 1-2: Facebook MAUs**



**Source: Facebook**



**Fig 1-3: If you printed twitter...**



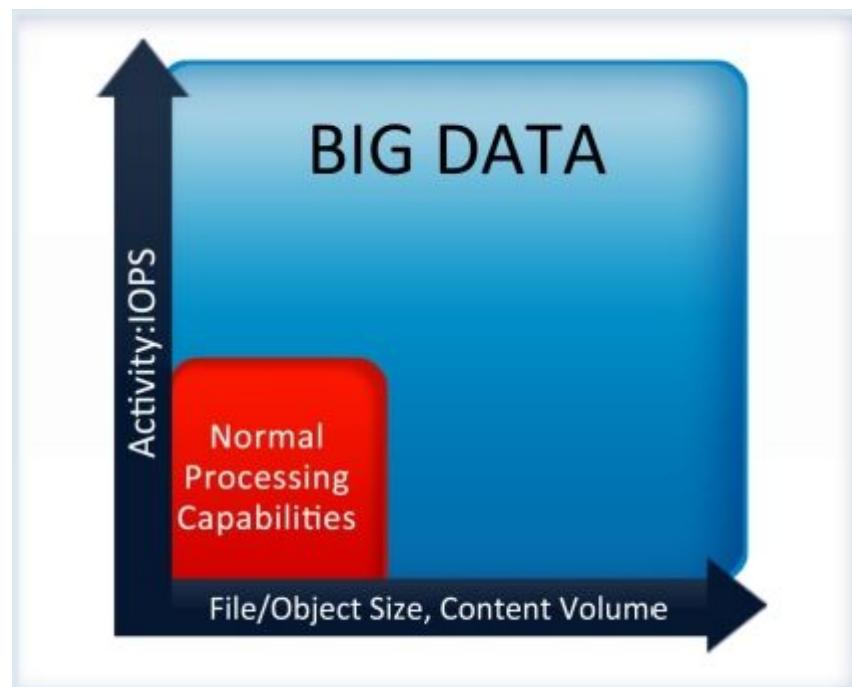
Just to explain with an example the amount of data which a single event of going and attending a Movie can generate .

*You may search for a movie on movie review sites, read reviews about that movie and post queries, you may tweet about the movie, post the photographs of going to the movie on Facebook . Maybe you also take a video of walking around the theater and upload it. While travelling your GPS system tracks your mobile and generates data. You may also use Google Latitude to update the status so that your friends can know where you are.*

*Thus a simple activity like above can generate enormous amounts of data.*

Hence if we have to summarize - the combination of smart phones, social networking sites, media and data internal to companies are creating flood of data for the companies to process and store.

 “*Big Data refers to data sets whose size is beyond the ability of typical database software tools to capture, store, manage and analyze.*” - *The McKinsey Global Institute, 2011*



*Fig 1-4: Big Data*

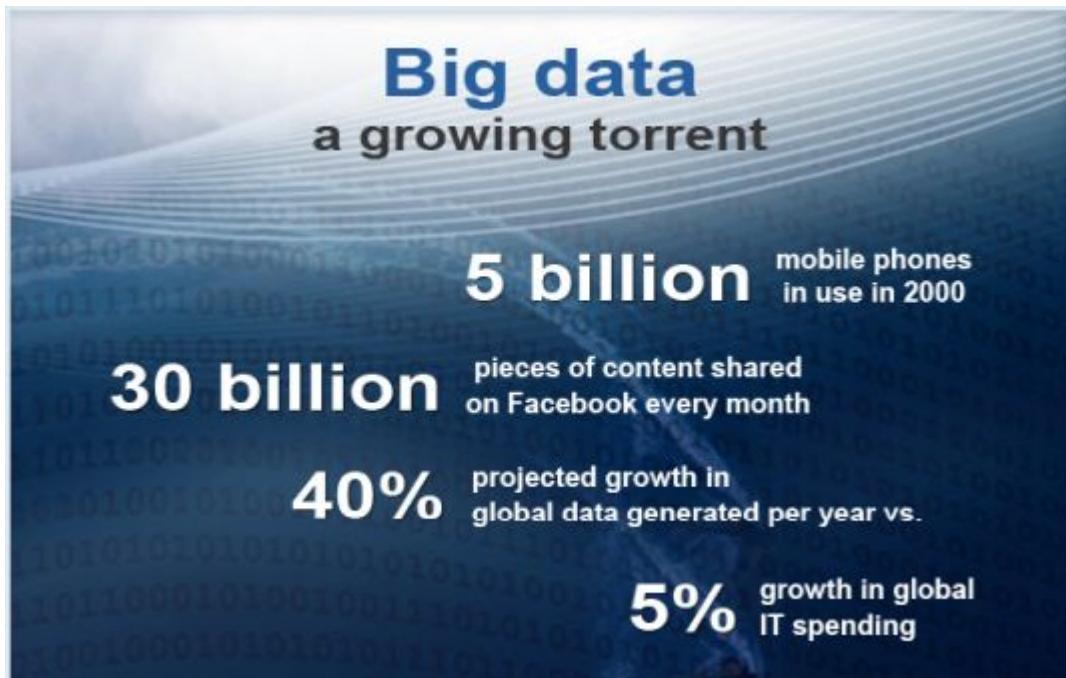
## BIG DATA

“Big Data is defined as data which has High Volume, Velocity and has multiple Varieties. “

Let's next look at few facts and figures of Big Data.

## Facts on Big Data

As we have seen in the previous section the rise of social media, multimedia and networked sensors are fuelling the exponential growth in data together with the increasing volume of data which is being captured by organizations now a days.



*Fig 1-5: Big Data – A growing torrent*

In this section we are going to discuss the trends on amount of data which is getting generated.

Various research teams in the world have done analysis on the amount of data being generated some of them are mentioned below for reference. One common theme which comes across is that the amount of data being generated and stored is increasing at an ever growing rate.

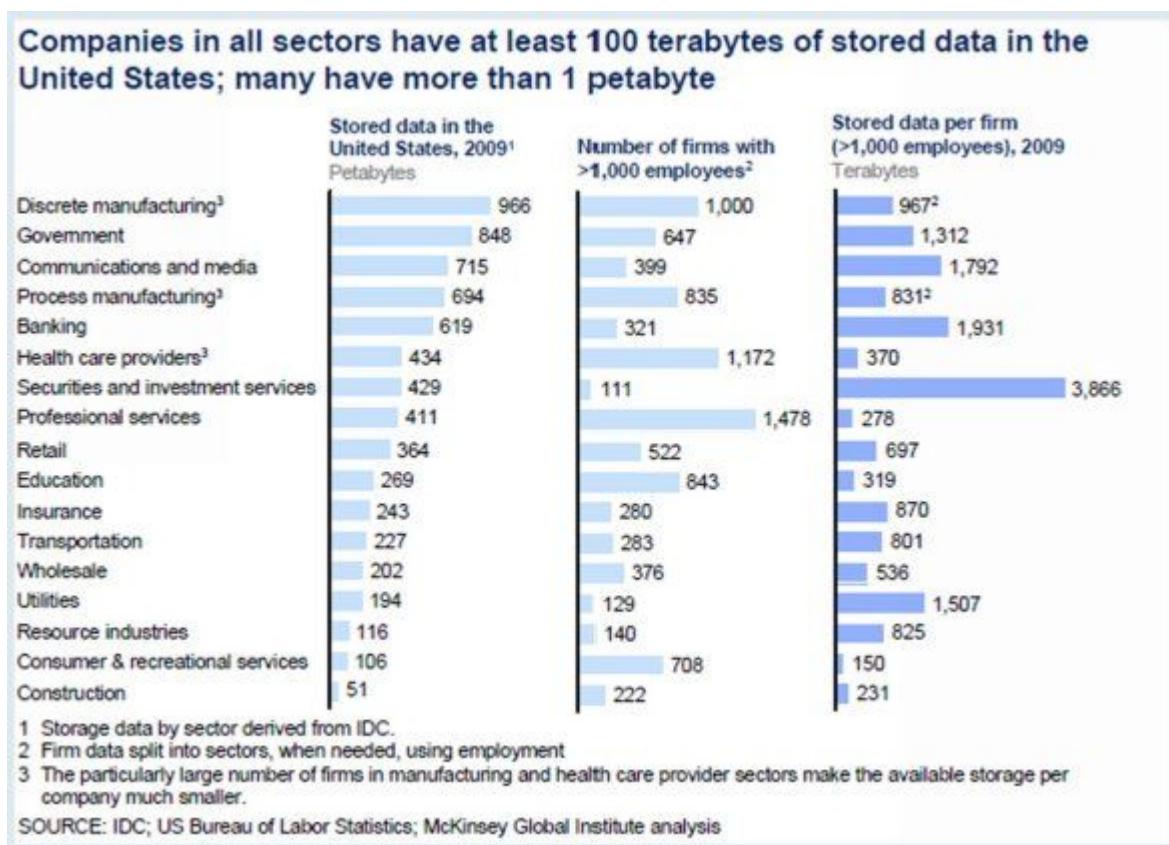
 *Report on enterprise server information, January 2011; and Martin Hilbert and Priscila López, "The world's technological capacity to store, communicate, and compute information," Science, February 10, 2011. IDC has published a series of white papers, sponsored by EMC, including "The expanding digital universe," March 2007; "The diverse and exploding digital universe," March 2008; "As the economy contracts, the digital universe expands," May 2009; and "The digital universe decade—Are you ready?," May 2010. All are available at [www.emc.com/leadership/programs/digital-universe.htm](http://www.emc.com/leadership/programs/digital-universe.htm).*

The IDC's analysis revealed that in year 2007, the amount of digital data generated in a year exceeded the world's total storage capacity which means there were no means by which all the data which is being generated can be stored. This also revealed the fact that the rate at which the data is getting generated will soon outgrow the rate at which the data storage capacity is expanding.

 *The following are citing's from the MGI (McKinsey Global Institute, established in 1990) report ([http://www.mckinsey.com/insights/business\\_technology/big\\_data\\_the\\_next\\_frontier\\_for\\_innovation](http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation)) published in May 2011.*

## **The Big Data Size Varies across sectors**

MGI estimates that enterprises around the world used more than 7 Exabyte's of incremental disk drive data storage capacity in 2010; nearly 80 percent of that total appeared to be duplicate data that had been stored elsewhere.



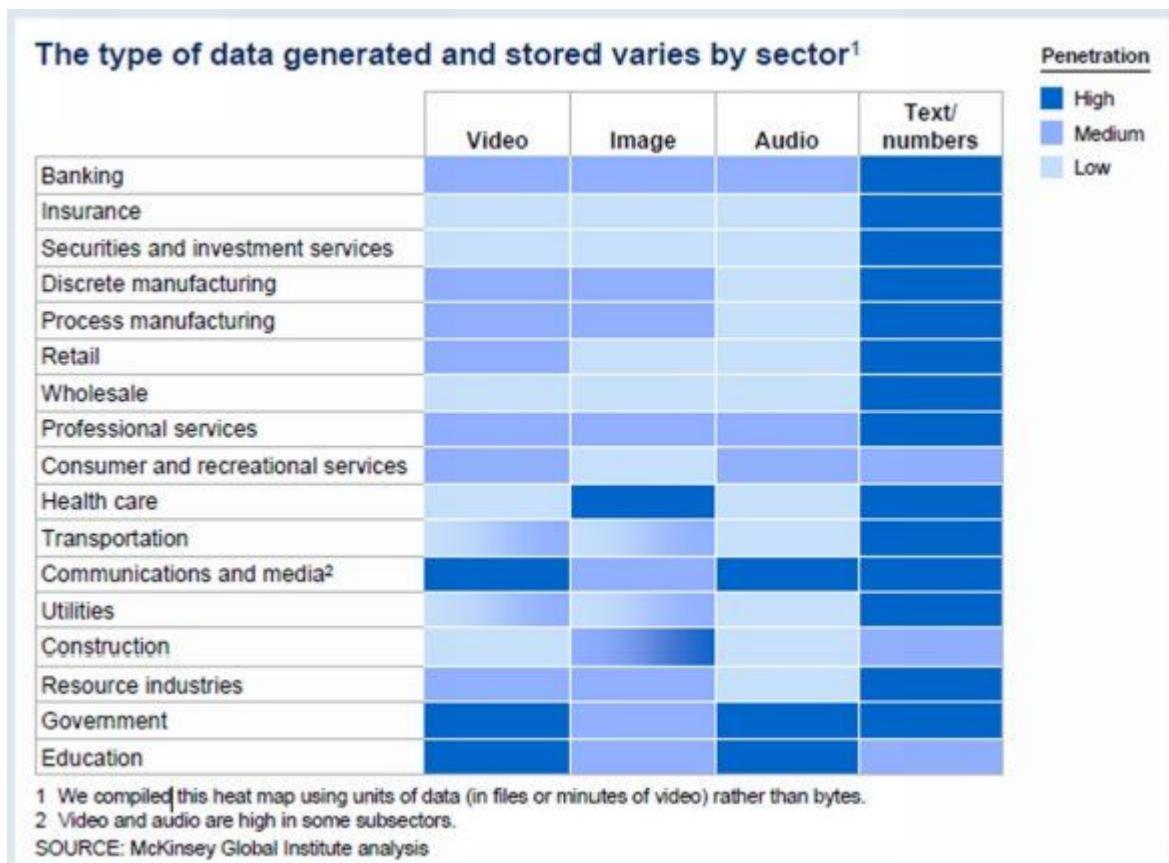
**Fig 1-6: Data Size variations Across Sectors**

As is obvious from the figure above some sectors are particularly heavy on data storage.

Financial Services are highly transaction oriented and also regulated to store data and thus the analysis shows them to be big users of data. Some of the businesses are highly dependent on creation and transfer of data e.g. media, communications etc. and thus figure in the list as high users of data storage.

## **The Big Data Type also Varies across sectors**

The MGI research shows that the type of data stored also varies by sector.



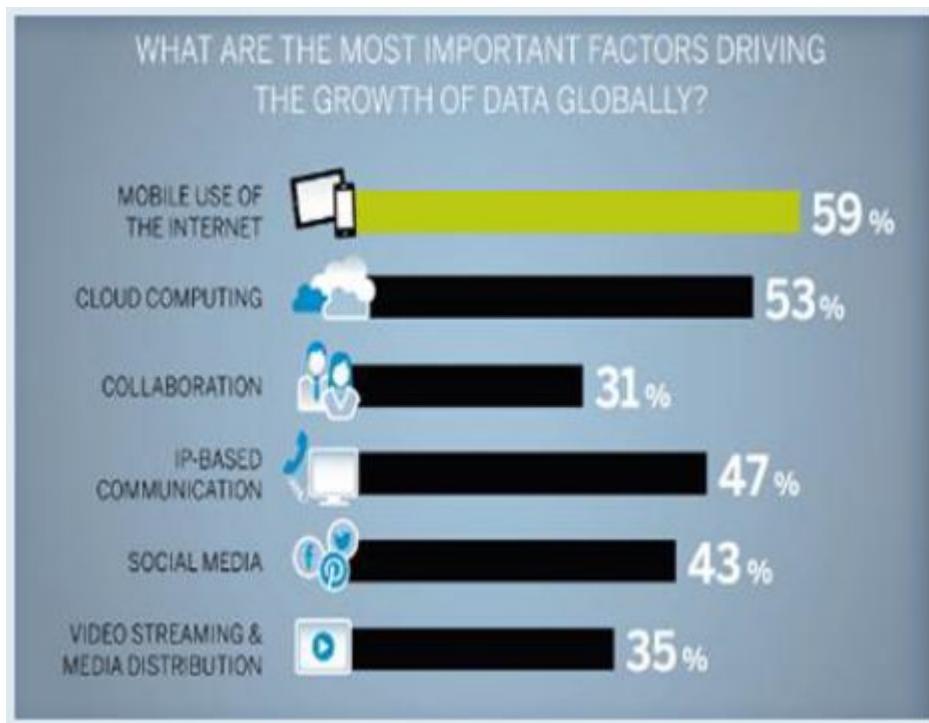
**Fig 1-7: Variety of Data across sectors**

In terms of geographic spread of big data, North America and Europe have 70% of the Global Total currently. With the advent of cloud computing data generated in one region can be stored in another country's datacenter thus countries with significant cloud and hosting provider presence tend to have high storage of data.

# Big Data Sources

In this section we will cover what are the major factors which are contributing to the ever increasing size of data. The below figure depicts few of the major contributing sources.

As can be seen the internet usage via mobile devices, popularity of social media sites, proliferation of networked sensors applications, expansion of multimedia contents are few of the major sources which are contributing to the ever increasing data.

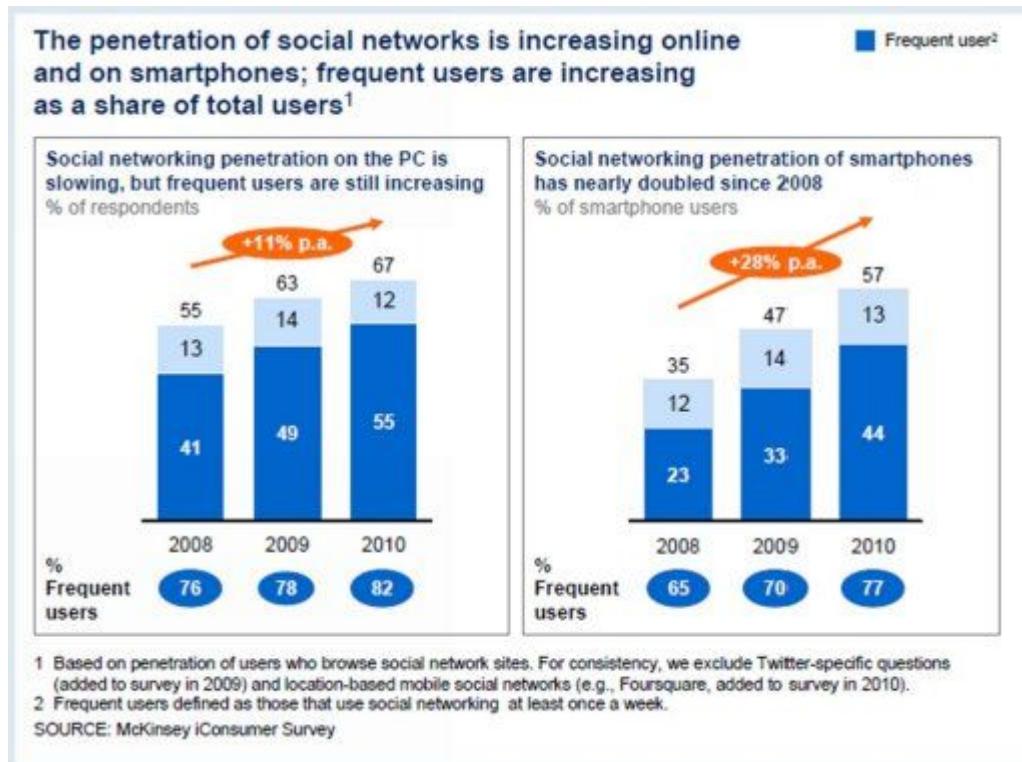


*Fig 1-8: Sources of Data*

As highlighted in the MGI report the major sources which are contributing to the ever increasing data set are

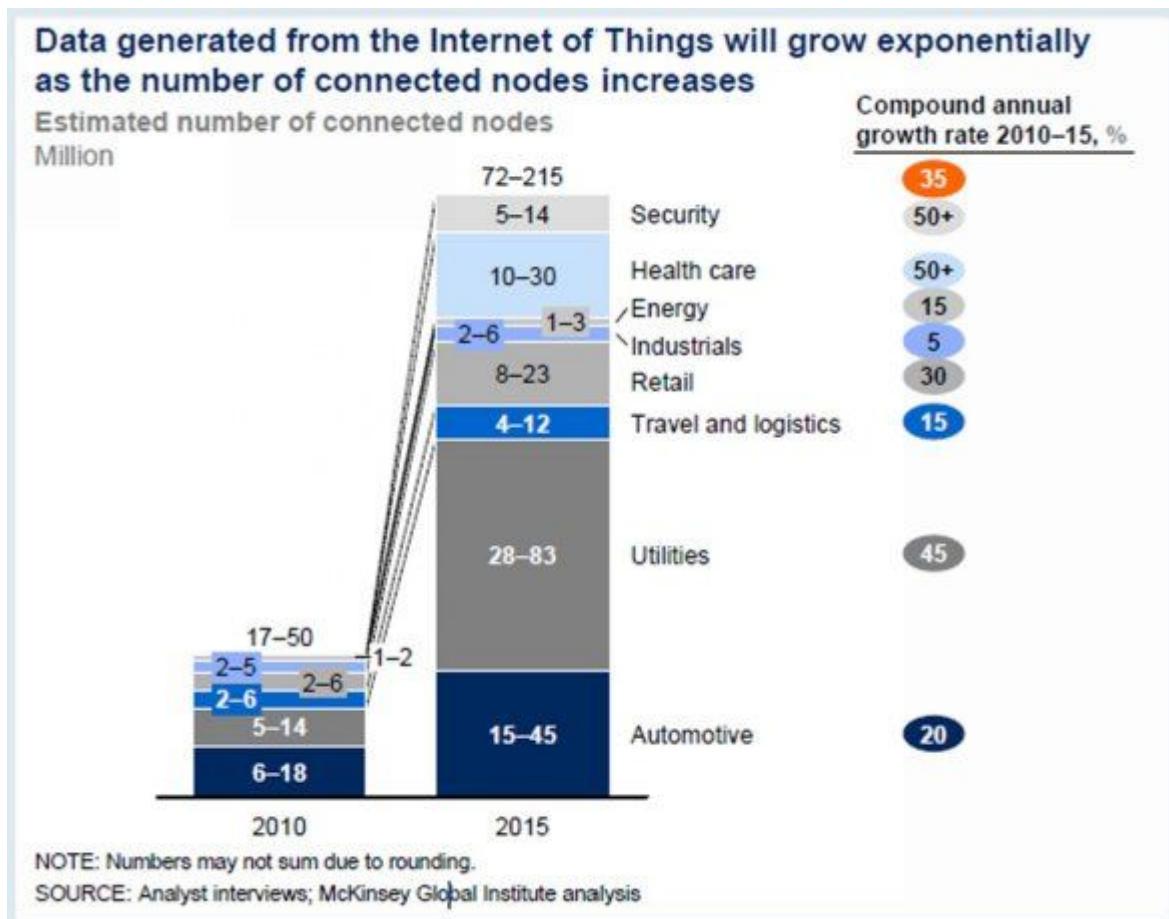
- 1: Enterprises which are collecting data with more granularities now, attaching more details with every transaction for understanding the consumer behavior.
- 2: Increased usage of multimedia across Sectors like health care, consumer facing industries etc.
- 3: Increased popularity of social media sites such as Facebook, twitter etc.
- 4: Rapid adoption of smart phones enabling the users to actively use these sites and other internet applications.
- 5: Increased usage of sensors and devices in day to day world which are connected by networks to computing resources.

**Fig 1-9:** Below figure is an illustration of Usage of Social Networks



**Source:** *McKinsey iConsumer Survey*

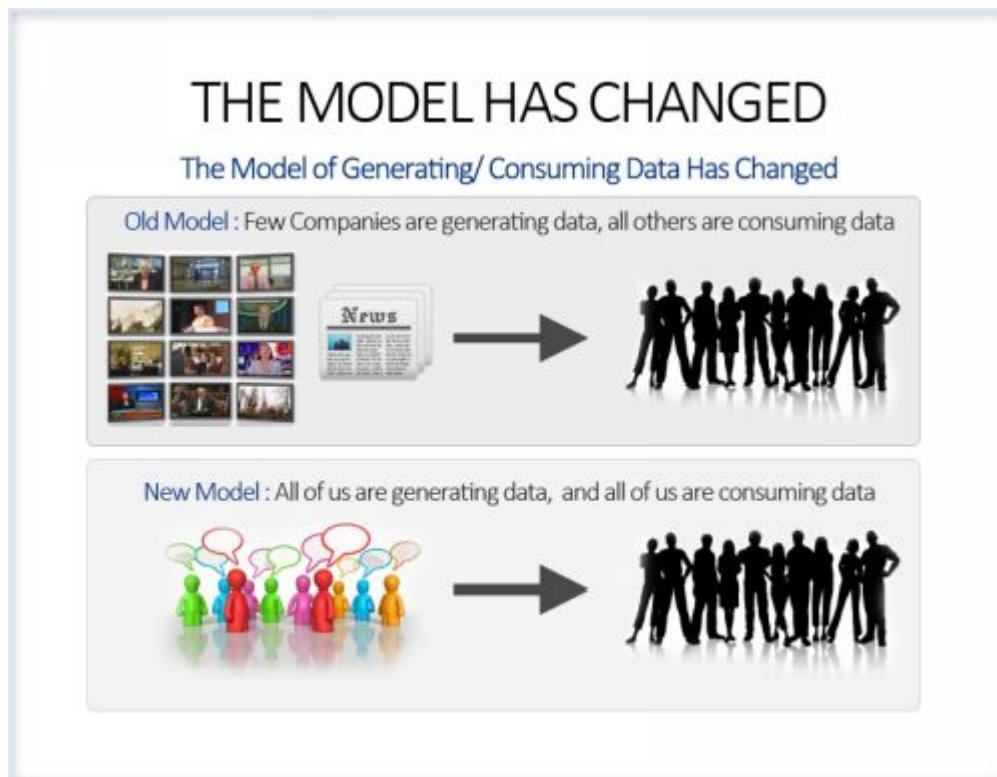
The MGI report also projects that the number of connected nodes in the Internet of Things—sometimes also referred to as machine-to-machine (M2M) devices—deployed in the world will grow at a rate exceeding 30 percent annually over the next five years.



**Fig 1-10: Data generation Report**

Hence as seen above the rate of growth of data is increasing and so is the diversity.

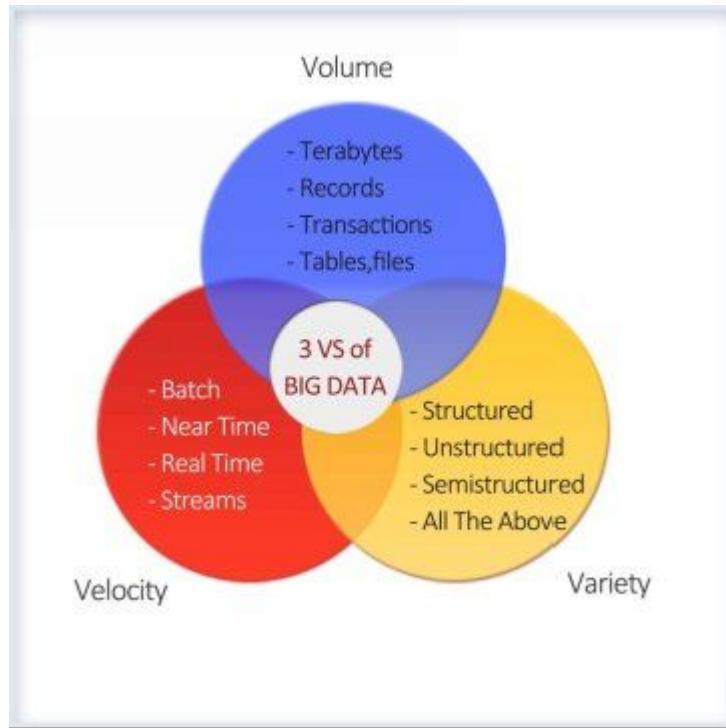
Also the model of data generation has changed from few companies generating data and others consuming it to everyone generating data and everyone consuming it. This as explained earlier is due to the penetration of Consumer IT and Internet technologies along with trends like social media.



***Fig 1-11: Data Model***

# Three V's of Big Data

We have defined the big data as data with 3 V's – Volume, Velocity and Variety.



**Fig 1-12: 3V's of Big Data**

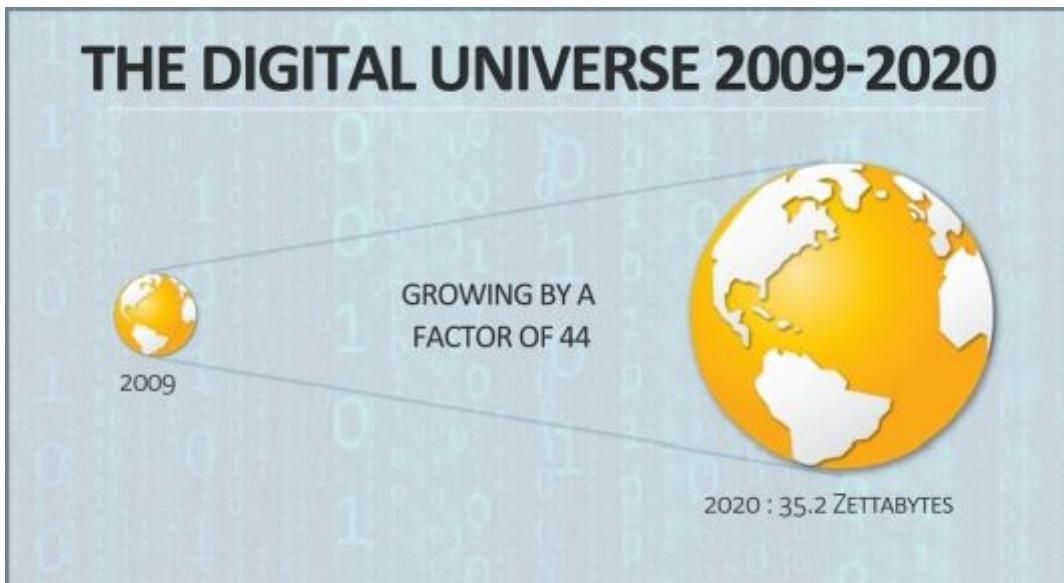


The “Big” in Big Data isn’t just about volume.

In this section we will look at the three V's. It is imperative that organizations and IT leaders focus on these 3 V's.

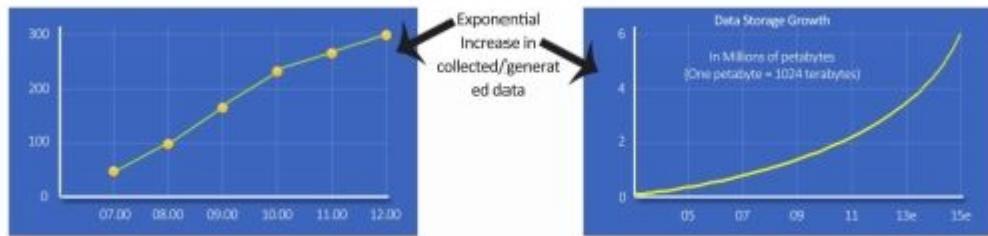
## Volume

Volume in Big Data means the size of the data. As discussed in the previous sections various factors contributes to the size of the Big Data – as businesses are becoming transaction oriented, we see ever increasing numbers of transactions, devices are getting connected to the internet which is adding to the amount of data being generated by these devices and tracked, increased usage of internet and digitization of content is resulting in increased volumes of data.



**Fig 1-13:** Digital Universe Size

In today's scenario data is not just generated from within the enterprise, but is generated based on transactions with the extended enterprise and the customers with the extensive maintenance of customer's data with the enterprises. In addition to this Machine data generated by smart phones, sensors etc. proliferation of social networking sites is also contributing to the growth. Petabyte scale data is becoming commonplace these days.



**Fig 1-14: Growth Rate**

This huge volume of data is the biggest challenge for Big Data technologies. The storage and processing power needed to store, process and make accessible the data in a timely and cost effective manner is the challenge.

## Variety

As we have seen in the previous sections with the proliferation of data sources from social sites, sensor devices, mobiles, smart phones different types of data are introduced. The data generated from various devices and sources follows no fixed format or structure. As compared to text, CSV or RDBMS data the data generated here varies from text files, log files, streaming videos, photos, meter readings, stock ticker data, PDFs, audio and various other unstructured formats.

There is no control over the structure of the data these days, new sources and structures of data are being created at a rapid pace. Thus the onus is on technology to find a solution to analyze and visualize the huge variety of data that is out there. As an example to provide alternate routes to commuters a traffic analysis application will need data feeds from millions of smartphones and sensors to provide accurate analytics on traffic conditions and alternate routes.



***Fig 1-15: Data Variety***

A variety of formats and unstructured data leads to a technology challenge to consume and analyze this disparate data.

## **Velocity**

Velocity in Big Data refers to the speed at which data is being generated and the speed at which it is required to be processed. If data cannot be processed at the required speed it loses its significance. Due to data streaming happening from the social media sites, sensors, tickers, metering, monitoring it is becoming important for the organizations to speedily process the data both when it's on move and when it's static. Reacting and processing quickly enough to deal with the velocity of data is one more challenge for the big data technology.



*Fig 1-16: Speed of Data*

Real time insight is essential in many big data use cases as an example Algorithmic Trading system take real time feeds from market and social media sites like twitter to take real time decisions on stock trading. Any delay in processing of this data can mean millions of dollars in lost opportunity on a stock trade.

**To summarize** Big Data is defined by three V's which are Volume, Velocity and Variety. The technology challenge in Big Data is the need for it to be processed within the available processing and storage resources in a cost effective manner.



*Fig 1-17: The three aspects of Data*

In addition to the 3 V's there's a fourth V which is talked about whenever Big Data is discussed. The fourth V is **veracity** which means not all the data out there is important hence it's very important to identify from this humongous data source as in what is important what will provide us meaningful insight and what should be ignored.

# Usage of big data

We have seen in the above sections what Big Data is, in this section we will focus on ways of using Big Data for creating value for the organizations.

Before we delve into how big data can be made usable to the organizations we will first look at why this big data is important.

Big data as we have seen above is a completely new source of data, data which is getting generated when you are posting on a blog, liking some product or moving. In older world such minutely available information was not captured, now this data is getting captured, hence organizations which embraces such data paves way for themselves for new innovations, in improving their agility and finally increasing their profitability.

Various ways are available in which the big data can create value for any organization.

As listed in the MGI report this can be broadly categorized into five ways of usage of big data:

## ***Visibility***

Making big data accessible in a timely fashion to relevant stakeholders creates tremendous amount of value.

Let's understand this with an example we have a manufacturing company which has the R&D, engineering and manufacturing departments dispersed geographically if the data is accessible across all these departments and can be readily integrated then it can not only reduce the search and processing time but will also help in improving the product quality in time according to the present needs.

## ***Discover and analyze information***

Most of the value of big data comes from when the data being collected from outside sources can be merged with the organization's internal data. Organizations are capturing detailed data be it of the inventories, employees or of the customer. Using all this data they can discover and analyze new information and patterns and hence this information and knowledge can be used to improve processes and performance.

## ***Segmentation and Customizations***

The big data enables organizations in creating highly specific segmentations and tailor products and services to meet there needs. This can also be used in the social sector to accurately segment populations and target benefit schemes targeted to specific needs.

Segmentation of customers based on various parameters can aid in targeted marketing campaigns and tailoring of products to suit the needs of customers.

## ***Aid Decision Making***

This data can substantially improve decision making, minimize risks, and unearth valuable insights that would otherwise remain hidden. Automated Fraud Alert systems in credit card processing to automatic fine tuning of inventory are examples of systems which aid or automate decision making based on Big Data Analytics.

## ***Innovation***

The big data enables innovation of new ideas in form of products and services. It enables innovation in the existing ones in order to reach out to large segments of people. Using data gathered for the actual products the manufacturers can not only innovate to generate the next generation product but they can also innovate there after sales offerings.

As an example real time data from machines and vehicles can be analyzed to provide insight into maintenance schedules, wear and tear of machines monitored to make resilient machines, fuel consumption monitoring can lead to higher efficiency engines. Real

time traffic information is already making life easier for commuters by providing them an option to take alternate routes.

Thus big data is not only about size but it's also about opportunities in finding meaningful insights from the ever increasing pool of data and helping the organizations in making more informed decisions making them more agile. Hence it not only provides the opportunity for organizations to strengthen their existing business by making informed decision it also helps them to expand their horizon.

# **BIG DATA Challenges**

Big Data also poses some challenges, in this section we will highlight a few of those:

## ***Policies and Procedures***

As more and more data is gathered, digitized and moved around the globe the policy and compliance issues become increasingly important. Data Privacy, Security, Intellectual property and protection of big data is of immense importance to organizations.

Compliance to various statutory and legal requirements poses a challenge in data handling. Issues around ownership and liabilities around data are important legal aspects which need to be dealt with in case of Big Data.

Also a lot of big data projects leverage the scalability features of public cloud computing providers. These two aspects combined together pose a challenge for Compliance.

Policy questions on who owns the data, what is defined as fair use of data, who is responsible for accuracy and confidentiality of data need to be answered.

## ***Access to data***

Accessing data for consumption is going to be a challenge for Big Data projects. Some of the data may be available with third parties and gaining access can be a legal, contractual challenge.

Data about a product or service is available on facebook, twitter feeds, reviews and blogs, how does the product owner access this data from various sources owned by various providers?

The contractual clauses and the economic incentives for accessing big data need to be tied in to enable the availability and access of data to the consumer.

## ***Technology and techniques***

Newer tools and technologies which are built specifically keeping in mind the needs of big data will have to be leveraged rather than trying to address these through legacy systems.

Thus the inadequacy of the legacy systems to deal with Big Data on one hand and the lack of experienced resources in newer technologies is a challenge that any big data project has to manage.

## **Legacy Systems and Big Data**

In this section we will discuss about the challenges that the organizations are facing for managing the big data using the legacy systems.

### ***Structure of Big Data***

The legacy systems are designed to work with structured data where tables with columns are defined. The Format of the data held in the columns is also known.

As opposed to this Big Data is Data with varied structures it's basically unstructured data such as images, videos, logs etc.

Since Big Data can be unstructured the legacy systems created to perform fast queries and analysis through techniques like indexing based on particular data types held in various columns cannot be used to hold or process Big Data.

### ***Data Storage***

The legacy system uses big servers and NAS and SAN system to store the data. As the data increases the server size and the backend storage size has to be increased. Traditional Legacy systems typically work in a scale up model where more and more compute, memory and storage needs to be added to a server to meet the increased data need.

Hence the time it takes to process the data increases exponentially hence defeating the other important requirement from Big Data i.e. Velocity.

## **Data Processing**

The algorithms which are designed in the legacy system are designed to work with structured data such as strings, integers and are also limited by the size of data.

Thus the legacy systems are not capable of handling the processing of unstructured data, huge volume of data and the speed at which the processing needs to be performed.

Hence to capture value from big data, new technologies (e.g., storage, computing, and analytical software) and techniques (i.e., new types of analyses algorithms and techniques) need to be deployed.

## **Big Data Technologies**

We have seen what big data is, in this section we will briefly look at what technologies can enable handling this humongous source of data. The technologies in discussion are not only needed to efficiently accept the data but they are also required to efficiently process different types of data in endurable time.

Hence a big data technology should not only be capable of collecting large amount of data but should also be able to process and analyze the data.

The recent technology advancements that enable organizations to make the most of its big data are

- 1: New Storage and processing technologies designed specifically for large unstructured data.
- 2: Parallel processing
- 3: Clustering
- 4: Large grid environments
- 5: High connectivity and high throughput
- 6: Cloud computing and scale out architectures.

There are a growing number of technologies which are making use of the technological advancements.

Here in this book we will be discussing about MongoDB one of the technologies which can be used to store and process big data.

## Summary

In this chapter we learned about Big Data. We looked into the various sources that are generating Big Data, the usage and challenges posed by Big Data. We also looked why newer technologies are needed to store and process Big Data.

In the following chapters we will look into few of the technologies which help organizations in managing Big Data and enables them to get meaningful insights from the BIG DATA.

# NOSQL

*“NoSQL is a new way of designing internet scale database solutions. It is not a product or technology but a term that defines a set of database technologies which are not based on the traditional RDBMS principles.”*

In this chapter we will cover the definition and basics of NOSQL. We will introduce the readers to the CAP theorem and will talk about the NRW notations. We will compare the ACID and BASE approaches and finally conclude the chapter by comparing NOSQL and SQL Database technologies.

# Introduction

## SQL

The idea of RDBMS was borne from E.F.Codd's whitepaper in 1970 titled "A relational model of data for large shared data banks". The language used to query RDBMS systems is SQL (Sequel Query Language).

RDBMS systems are well suited for structured data held in columns and rows which can be queried using SQL. The RDBMS systems are based on the concept of "ACID" transactions.

**ACID** stands for Atomic, Consistent, Isolated and Durable where:

- Atomic means that in a transaction either all the changes are applied completely or not applied at all.
- Consistent means the data is in a consistent state after the transaction is applied which means after a transaction is committed the queries fetching a particular data will see the same result.
- Isolated means the transactions that are applied to the same set of data are independent of each other. Thus one transaction will not interfere with the other transaction.
- Finally durable means the changes are permanent in the system and will not be lost in case of any failures.

## NOSQL

NoSQL is a term used to define non-relational databases. Thus it encompasses majority of the data stores which are not based on conventional RDBMS principles for handling large data sets on an internet scale.

Big data as discussed in the previous chapter is posing challenges to the traditional ways of storing and processing data i.e. the RDBMS

systems. Thus we see the rise of NOSQL databases which are designed to process this huge amount and variety of data within the time and cost constraints.

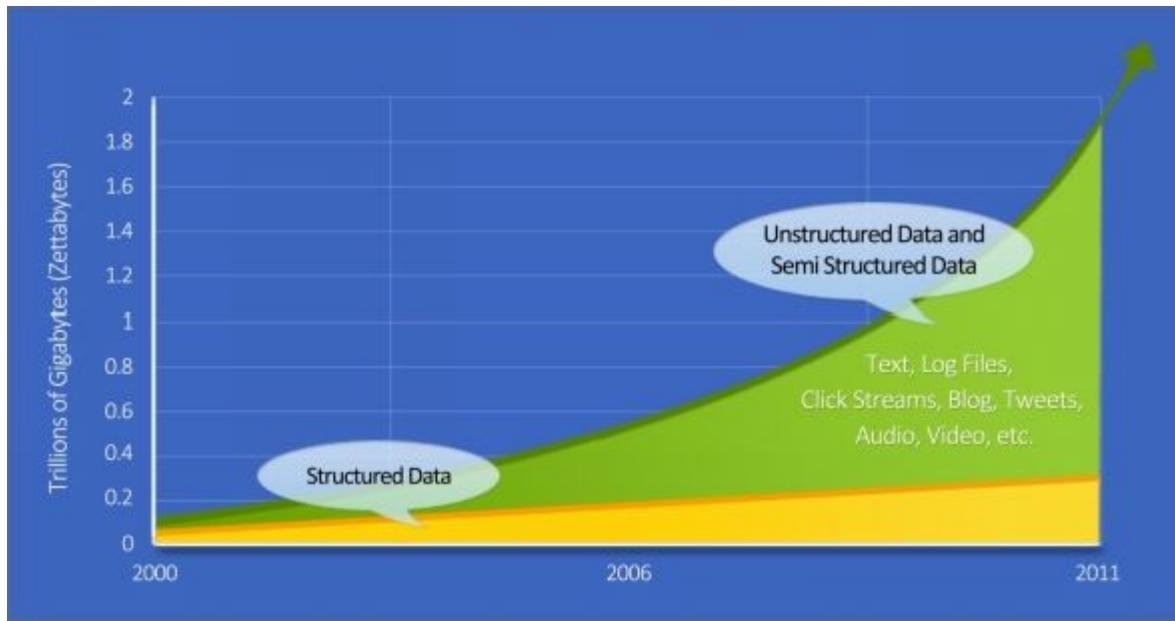
Thus NoSQL databases evolved from the need of handling Big Data where the traditional RDBMS technologies did not provide adequate solutions.



*Some examples of Big Data use cases which NOSQL Databases can meet are:*

*Social Network Graph: Who is connected to whom? Whose post should be visible on the user's wall or homepage on a social-network site.*

*Search and Retrieve: Search all relevant pages with a particular keyword ranked by the number of times a keyword appears on a page.*



*Fig 2-1: Structured Vs. Un/Semi-Structured Data*

## Definition

NoSQL doesn't have a formal definition. It represents a form of persistence/data storage mechanism that is fundamentally different from RDBMS. Hence if we have to define NoSQL, it will be as follows NoSQL is an umbrella term for data stores that don't follow the RDBMS principles.

*The term was used initially to mean "do not use SQL if you want to scale" later this was redefined to "not only SQL" which means that in addition to SQL other complimentary database solutions also exist.*

## Brief history of NoSQL

In the year 1998 Carlo Strozzi coined the term NoSQL to refer to his open source, light weight database. He used this term to identify his database because the database didn't have a SQL interface. The term resurfaced in early 2009 when Eric Evans (a Rackspace employee) used this term in an event on open source distributed databases to refer to the distributed databases which were non-relational and does not follow the ACID feature of the relational databases.

## ACID vs. BASE

In the introduction we referred that the traditional RDBMS applications have focused on ACID transactions.

Howsoever essential these qualities may seem, they are quite incompatible with availability and performance requirements for applications of web-scale.

*Let's say for example, we have a company like OLX which is used for selling products such as unused house hold goods for e.g. old furniture, vehicles etc. with RDBMS as its database.*



*Let's consider two scenarios*

*First Scenario: Let's look at an ecommerce shopping site where a user is buying a product. Now this may lock the database for updation when the user is buying a product so that the inventory is correctly reflected. In this case the user will end up locking a part*

*of database which is the inventory and every other user will end up waiting until the user who has put the lock completes the transaction. In another scenario the application might end up using cached data or even unlocked records resulting in inconsistency. In this case two users might end up buying the product when the inventory actually was zero.*

*The system may become slow impacting scalability and user experience.*

Now let us look at how NOSQL tries to solve this problem using an approach popularly called as “BASE”. Before explaining BASE, let’s understand the concept of CAP theorem.

## CAP Theorem (Brewer's Theorem)

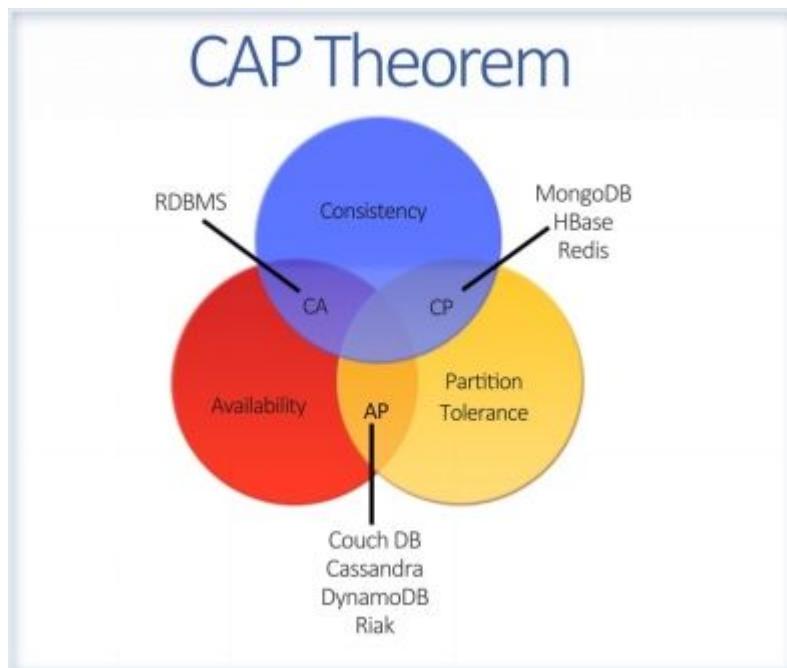
Eric Brewer outlined the CAP theorem in 2000. This is one of the important concepts that need to be well understood by developers and architects dealing with distributed databases. The theorem states when designing an application in a distributed environment there are three basic requirements which exist namely Consistency, Availability and Partition tolerance where

**Consistency** means the data remains consistent after any operation is performed that changes the data and all users or clients accessing the application sees the same updated data.

**Availability** means the system is always available with no downtime.

**Partition Tolerance** means the system will continue to function even if it’s partitioned into groups of servers which are not able to communicate with one another.

The CAP theorem states that at any point of time a distributed system can fulfill only two of the above three guarantees.



*Fig 2-2: CAP Theorem*

## The BASE

Eric Brewer coined the BASE acronym.

BASE can be explained as:

- **Basically Available** means the system does guarantee availability, in terms of the CAP theorem.
- **Soft state** indicates the system state may change over time even if no input is there. This is in compliance to the eventual consistency model.
- **Eventual consistency** indicates that the system will become consistent over a period of time provided no input is sent to the system during that time.

Hence BASE is in contrast with the RDBMS ACID transactions.

We have seen that NoSQL databases are eventually consistent but the eventual consistency implementation may vary across different NoSQL databases.

NRW is the notation which is used to describe how the eventual consistency model is implemented across NOSQL databases where

- N is the number of data copy which the database has maintained.
- R is the number of copy which an application needs to refer to before returning a read request's output.
- W is the number of data copies that need to be written to before a write operation is marked as completed successfully.

Hence using these notation configurations the databases implement the model of eventual consistency.

*Consistency can be implemented at both read and write operation level.*

### *Write Operations*

*N=W implies that the write operation will update all data copies before returning the control to the client and marking the write operation as successful. This is similar to how the traditional RDBMS databases work when implementing synchronous replication. This setting will slow down the write performance.*

*If write performance is a concern which means you want the writes to be happening fast then in that case we can set W=1, R=N which implies that the write will just update any one copy and mark the write as successful, but whenever the user issues a read request it will read all the copies to return the result. If either of the copy is not updated it will ensure the same is updated and then only the read will be successful but this implementation will slow down the read performance.*

*Hence most NoSQL implementations uses N>W>1 which implies that more than one writes must complete, but not all nodes need to be updated immediately.*

### *Read Operations*

*If R is set to 1 then the read operation will read any data copy which can be out dated. If R>1 then more than one copy is read and hence will read most recent value but this can slow down the read operation.*

*Using W+R>N ensures a read always retrieves the latest value. Reason being the number of copies which is written and number of copies read is high enough to guarantee that always at least one copy of the latest version is read in the read set. This is referred as **quorum assembly**.*

**Table 2-1:** Table depicts NRW configuration

Configuration	Outcome
<b>W=N R=1</b>	Read optimized strong consistency
<b>W=1 R=N</b>	Write optimized strong consistency
<b>W+R&lt;=N</b>	Weak eventual consistency. A read might not see latest update.
<b>W+R&gt;N</b>	Strong consistency through quorum assembly. A read will see at least one copy of the most recent update.

Hence if we have to compare ACID vs. BASE it'll appear as below

**Table 2-2:** ACID vs. BASE

ACID	BASE
<b>Atomicity</b>	Basically Available
<b>Consistency</b>	Eventually Consistency
<b>Isolation</b>	Soft State
<b>Durable</b>	

## NoSQL Advantages and Disadvantages

In this section we will look at the advantages and disadvantages NoSQL databases.

### Advantages of NoSQL

**High scalability**

This scaling up approach fails when the transaction rates and the fast response requirement increase. In contrast to this the new generation of NoSQL databases were designed to scale out i.e. expand horizontally using low end commodity servers.

### **Manageability and Administration**

NoSQL databases are designed to mostly work with automated repairs, distributed data and simpler data models hence leading to low manageability and administration.

### **Low Cost**

The NoSQL databases are typically designed to work with cluster of cheap commodity servers enabling the users to store and process more data at a low cost.

### **Flexible data models**

NoSQL databases have a very flexible data model enabling them to work with any type of data; they don't comply with the rigid RDBMS data models. Hence any application changes which involve updating the database schema can be easily implemented.

# Challenges of NoSQL

In addition to the above mentioned advantages there are many impediments that users need to be aware of before they start developing applications using these platforms.

## Maturity

Most of the NoSQL databases are still in pre-production versions with key features that are still to be implemented, hence while deciding on a NoSQL database organizations should analyze the product properly to ensure there requirement is fully implemented and is not in the To-do list.

## Support

Support is one limitation users' needs to be aware of. As most of the NoSQL's are open source projects hence there are one or more firms offering support for the databases, generally these are small startups and as compared to the enterprise software companies and may not have global reach or support resources.

## Limited Query Capabilities

As NoSQL databases were generally developed to meet the scaling requirement of the web-scale applications hence they provide limited querying capabilities. A simple querying requirement may involve significant programming expertise.

## Administration

Though NoSQL is designed to provide no-admin solution however it still requires skill and effort for installing and maintaining the solution.

## Expertise

Since NoSQL is an evolving area hence expertise of the technology is limited in the developer and administrator community.

Though NoSQL is becoming an important part of the database landscape but the users need to be aware of the limitations and

advantages of the products to make a correct choice of the NoSQL database platform.

## SQL vs. NoSQL Databases

We have seen what NoSQL databases are. Though it's increasingly getting adopted as a database solution but it's not here to replace SQL or RDBMS databases. In this section we will look at the differences between SQL and NoSQL databases.



*Fig 2-3: NoSQL World*

Let's do a quick recap of the RDBMS system. The RDBMS systems have prevailed for about 30 years and even now it's a default choice for the solution architect for data storage for an application. If we will list down few of the good points of the system - the first and the foremost is the use of SQL which is a rich declarative query language used for data processing, its well understood by the users and in addition they have ACID support for transaction which is a must in many of the applications such as banking applications.

However the biggest drawback of the RDBMS system is difficulty in handling schema changes, scaling issues as the data increases. As the data increases the read read/write performance degrades. We face scaling issues with RDBMS systems as they are mostly designed to Scale Up and not Scale Out.

In contrast to the SQL RDBMS databases NoSQL promotes the data stores which break away from the RDBMS paradigm.

*Let's talk about technical scenarios and how they compare in RDBMS Vs NoSQL:*

**Schema flexibility:** This is a must for easy future enhancements and integration with external applications (outbound or inbound).

RDBMS are quite inflexible in their design. More often than not, adding a column is an absolute no-no, especially if the table has some data. The reasons range from default value, indexes, and performance implications. More often than not we end up creating new tables and increase complexity by introducing relationships across the tables.

**Complex queries:** The traditional designing of the tables leads to developers ending up writing complex JOIN queries which are not only difficult to implement and maintain but also take substantial database resources to execute

**Data update:** Updating data across tables is probably one of the more complex scenarios especially if they are a part of the transaction. Note that keeping the transaction open for a long duration hampers the performance. One also has to plan for propagating the updates to multiple nodes across the system. And if the system does not support multiple masters or writing to multiple nodes simultaneously, there is a risk of node-failure and the entire application moving to read-only mode.

**Scalability:** More often than not, the only scalability that may be required is for read operations. However, several factors impact this speed as operations grow. Some of the key questions to ask are:

- *What is the time taken to synchronize the data across physical database instances?*
- *What is the time taken to synchronize the data across datacenters?*
- *What is the bandwidth requirement to synchronize data? Is the data exchanged optimized?*
- *What is the latency when any update is synchronized across servers? Typically, the records will be locked during an update.*

**NoSQL-based solutions** provide answers to most of the challenges listed above. Let us now see what NoSQL has to offer against each technical question mentioned above:

**Schema flexibility:** Column-oriented databases store data as columns as opposed to rows in RDBMS. This allows flexibility of adding one or more columns as required, on the fly. Similarly, document stores that allow storing semi-structured data are also good options.

**Complex queries:** NoSQL databases do not have support for relationships or foreign keys. There are no complex queries. There are no JOIN statements.

*Is that a drawback? How does one query across tables?*

*It is a functional drawback, definitely. To query across tables, multiple queries must be executed. Database is a shared resource, used across application servers and must not be released from use as quickly as possible. The options involve combination of simplifying queries to be executed, caching data, and performing complex operations in application tier. A lot of databases provide in-built entity-level caching. This means that as and when a record is accessed, it may be automatically cached transparently by the database. The cache may be in-memory distributed cache for performance and scale.*

**Data update:** Data update and synchronization across physical instances are difficult engineering problems to solve.

*Synchronization across nodes within a datacenter has a different set of requirements as compared to synchronizing across multiple datacenters. One would want the latency within a couple of milliseconds or tens of milliseconds at the best. NoSQL solutions offer great synchronization options.*

*MongoDB, for example, allows concurrent updates across nodes, synchronization with conflict resolution and eventually, consistency across the datacenters within an acceptable time that would run in few milliseconds. As such, MongoDB has no concept of isolation. Note that now because the complexity of managing the transaction may be moved out of the database, the application will have to do some hard work.*

*An example of this is a two-phase commit while implementing transactions*

*(<http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/>).*

*A plethora of databases offer Multiversion concurrency control (MCC) to achieve transactional consistency ([http://en.wikipedia.org/wiki/Multiversion\\_concurrency\\_control](http://en.wikipedia.org/wiki/Multiversion_concurrency_control)).*

*Surprisingly, eBay does not use transactions at all (<http://www.infoq.com/interviews/dan-pritchett-ebay-architecture>).*

*Well, as Dan Pritchett (<http://www.addsimplicity.com/>), Technical Fellow at eBay puts it, eBay.com does not use transactions. Note that PayPal does use transactions.*

**Scalability:** NoSQL solutions provide greater scalability for obvious reasons. A lot of complexity that is required for transaction oriented RDBMS does not exist in ACID non-compliant NoSQL databases. Interestingly, since NoSQL do not provide cross-table references and there are no JOIN queries possible, and because one cannot write a single query to collate data across multiple tables, one simple and logical solution is to—at times—duplicate the data across tables. In some scenarios, embedding the information within the primary entity—especially in one-to-one mapping cases—may be a great idea.

Let's compare SQL and NOSQL technologies:

**Table 2-3: SQL vs. NoSQL**

	<b>SQL Databases</b>	<b>NoSQL Databases</b>
<b>Types</b>	All types supports SQL standard.	Different Types of databases exists such as Key Value stores, Document stores, Column Databases etc.
<b>Development History</b>	Developed in 1970	Developed in 2000s
<b>Examples</b>	SQL Server, Oracle, MySQL	MongoDB, HBase, Cassandra
<b>Data Storage Model</b>	<p>Data is stored as row and column in a table. Where each column is of specific type.</p> <p>The tables generally are created on principles of Normalization.</p> <p>Joins are used to retrieve data from multiple tables.</p>	<p>The data model depends on the database type. Say for e.g. data is stored as a key, value pair for key value stores. For documents based databases the data is stored as documents.</p> <p>The data model is flexible in contrast to the rigid table model of the RDBMS.</p>
<b>Schemas</b>	Fixed structure and schema, thus any change to schema involves altering the database.	Dynamic Schema, new data types or structures can be accommodated by expanding or altering the current schema. New fields can be added dynamically.
<b>Scalability</b>	Scale Up Approach is used means as the load increases bigger expensive servers are bought to accommodate the data.	Scale Out Approach is used means distributing the data load across inexpensive commodity servers.

	SQL Databases	NoSQL Databases
<b>Supports Transactions</b>	Supports ACID and Transactions	Supports partitioning and availability and compromises on Transactions. Transactions exist at certain level e.g. database level or document level.
<b>Consistency</b>	Strong consistency	Dependent on the product. Few chose to provide strong consistency whereas few provide eventual consistency.
<b>Support</b>	High level of enterprise support is provided.	Open source model. Support through third parties or companies building the open source products.
<b>Maturity</b>	Have been around for a long time	Some of them are mature others are evolving.
<b>Querying Capabilities</b>	Available through easy to use GUI interfaces.	Querying may require programming expertise and knowledge. Rather than UI focus is on functionality and programming interfaces.
<b>Expertise</b>	Large community of developers who have been leveraging the SQL language and RDBMS concepts to architect and develop applications.	Small community of developers working on these open source tools.

# Categories of NoSQL database

In this section we will quickly analyze the NoSQL Landscape. We will look at the emerging categories of NoSQL databases. The following table is a listing of few of the projects in the NoSQL landscape with the types and the players in each category.

The NoSQL databases are categorized on the basis of how the data is stored. NoSQL mostly follows a horizontal structure because of the need to provide curated information from large volumes, generally in near real-time. They are optimized for insert and retrieve operations on a large scale with built-in capabilities for replication and clustering.

**Table 2-4: NoSQL categories**

Category	Description	Name of the database
<b>Document Oriented</b>	Stores data in form of documents. For e.g. {Name="Test User", Address="Address1", Age:8}	MongoDB, CouchDB, SimpleDB etc.
<b>XML database</b>	Stores data in XML format	MarkLogic, etc.
<b>Graph databases</b>	Stores data as collection of nodes which are connected using edges. Nodes are comparable to objects in a programming language.	Neo4j, GraphDB etc.
<b>Key-value store</b>	Stores data as Key-Value pairs.	Cassandra, Redis, memcached etc.

The following table briefly provides a feature wise comparison between the various categories of NoSQL databases

**Table 2.5: Feature wise comparison**

Feature	Column Oriented	Document Store	Key Value Store	Graph
<b>Table Like Schema Support (Columns)</b>	Yes	No	No	Yes
<b>Complete Update/Fetch</b>	Yes	Yes	Yes	Yes
<b>Partial Update/Fetch</b>	Yes	Yes	Yes	No
<b>Query/Filter on Value</b>	Yes	Yes	No	Yes
<b>Aggregate Across Rows</b>	Yes	No	No	No
<b>Relationship between entities</b>	No	No	No	Yes
<b>Cross-entity view support</b>	No	Yes	No	No
<b>Batch Fetch</b>	Yes	Yes	Yes	Yes
<b>Batch Update</b>	Yes	Yes	Yes	No

 *The important thing while considering a NoSQL project is the feature set the users are interested in.*

Hence when deciding on the NoSQL product first we need to understand the problem requirement very carefully, then we should look at other people who have already started using the NoSQL product in solving similar problems. This is done because the NoSQL is still maturing hence this will enable us to learn from the peers and previous deployments and make better choices.

In addition we also need to consider the following - How big is the data that need to be handled, what throughput is acceptable for read and write, how consistency is achieved in the system, whether the system needs to support high write performance, or high read performance, how easy is the maintainability and administration, what needs to be queried, what is the benefit of using NoSQL. And finally start small but significant and consider a hybrid approach wherever possible.

# Summary

In this chapter we learned about NoSQL. We understood what NoSQL is, how is it different from SQL. We also looked into the various categories of NoSQL.

In the following chapters we will look into MongoDB which is a Document based NoSQL database.

# Introducing MongoDB

*“MongoDB is one of the leading NoSQL document store databases which enable organizations to handle and gain meaningful insights from Big Data.”*

In this chapter we will introduce you to MongoDB. We will look at the design decisions leading to the product MongoDB and will also let the readers know about the origination of the MongoDB product.

## Introduction

MongoDB is one of the leading NoSQL document store platform which enables organizations to handle Big Data. MongoDB is a platform of choice for some of the leading enterprises and consumer IT companies who have leveraged the scaling and Geospatial Indexing capabilities of MongoDB in their products and solutions.

MongoDB derives its name from the word “*Humungous*”. Like other NoSQL databases MongoDB also doesn’t comply with the RDBMS principles. It doesn’t have concepts of tables, rows, columns it doesn’t provide features of ACID compliance, JOINS, foreign keys etc.

In this chapter we will talk a bit about MongoDB’s origination and the design decisions, we will look at the key features, components and architecture of MongoDB in the following chapters.

## History

In the later part of 2007 Dwight Merriman and his team (including Eliot Horowitz, ShopWiki founder) set out to create an online service for developing, hosting and automatically scaling web applications (in line with Google App engine or Microsoft Azure). Soon they realized no open source database platform suited the requirement of the service.

*Merriman says. "We felt like a lot of existing databases didn't really have the 'cloud computing' principles you want them to have: elasticity, scalability, and ... easy administration, but also ease of use for developers and operators," "[MySQL] doesn't have all those properties."*

([http://www.theregister.co.uk/2011/05/25/the\\_once\\_and\\_future\\_mongodb/](http://www.theregister.co.uk/2011/05/25/the_once_and_future_mongodb/))

Hence they decided to build a database that will not comply with the RDBMS model. After a year of work the database for the service was ready to use, the service never released but the team decided to open source the database as MongoDB. MongoDB was built under the aegis of the New York-based startup [10gen](#).

In 2009 MongoDB was open-sourced as a stand-alone product (1). Around March 25, 2010 with the release of MongoDB 1.4.0 it has been considered Production ready (2).



The latest production release is 2.4.9 and was released in January 2014.

(1)<http://blog.mongodb.org/post/103832439/the-agpl>

(2)<http://blog.mongodb.org/post/472835820/mongodb-1-4-ready-for-production>

## Definition

*"MongoDB is a Leading NoSQL open-source document based database system developed and supported by 10gen. It's provides high performance, high availability and easy scalability."*

MongoDB stores data as Binary JSON Documents (also known as BSON). The documents can have different schemas hence enabling

the schema to change as the application evolves.

MongoDB is built for scalability, performance and high-availability.

## MongoDB Design Philosophy

*MongoDB wasn't designed in a lab. We built MongoDB from our own experiences building large scale, high availability, and robust systems. We didn't start from scratch; we really tried to figure out what was broken, and tackle that. So the way I think about MongoDB is that if you take MySql, and change the data model from relational to document based, you get a lot of great features: embedded docs for speed, manageability, agile development with schema-less databases, easier horizontal scalability because joins aren't as important. There are lots of things that work great in relational databases: indexes, dynamic queries and updates to name a few, and we haven't changed much there. For example, the way you design your indexes in MongoDB should be exactly the way you do it in MySql or Oracle, you just have the option of indexing an embedded field.*

—Eliot Horowitz, 10gen CTO and Co-founder

In this section we will briefly look at some of the design decisions that led to what MongoDB is today.

## Speed, Scalability and Agility

The design team's goal while designing MongoDB was to create a database which is fast, massively scalable and is easy to use. To achieve speed and horizontal scalability in a partitioned database as explained in the Brewer's CAP theorem the Consistency and Transactional support have to be compromised. Thus as per the Brewer's CAP theorem MongoDB provides High Availability, Scalability and Partitioning at the cost of Consistency and Transactional support.

Thus MongoDB is a platform of choice for applications needing a flexible schema, speed and partitioning capabilities while it may not

be suited for applications which require consistency and atomicity.

Instead of tables and rows MongoDB uses documents to make it flexible, scalable and fast.

# Non-Relational Approach

Traditional RDBMS platforms provide scalability using scale up approach. One needs to provide a faster server to increase performance.

The following issues in RDBMS led to why MongoDB and other NoSQL databases were designed the way they are designed:

In order to scale out the RDBMS database needs to link the data available in two or more systems in order to report back the result. This is difficult to achieve in RDBMS's since they are designed to work when all the data is available for computation together. Thus the data has to be available for processing at a single location.

In case of multiple Active-Active servers when both are getting updated from multiple sources there is a challenge in determining which update is correct.

In case an application tries to read data from the second server whereas the information has been updated on the first server but yet to be synchronized with the second server, the information returned may be stale.

Hence MongoDB team decided to take a non-relational approach to solving this problem and providing a scalable, high performance database.

As we have mentioned above MongoDB stores its data in BSON documents where all the related data is placed together which means everything which is needed is at one place. The queries in MongoDB are based on keys in the document; hence the documents can be spread across multiple servers. While querying each server will check its own set of documents and return the result. This enables liner scalability and improved performance.

MongoDB has a master-slave replication where only one master exists which accepts the write requests. If the write performance need to be improved then "Sharding" can be used which splits the

data across multiple machines and hence enabling multiple machines updates different parts of the datasets. Sharding is automatic in MongoDB, as more machines are added data is distributed automatically.

We will be discussing the replication, sharding, data storage in detail in the subsequent sections.

# Transactional Support

In order to make MongoDB scale horizontally the transactional support had to be compromised.

This is an important design consideration for many NoSQL databases. We had discussed this aspect in the CAP theorem that a distributed database system can only cater to two of the three parameters thus to provide Availability and Partitioning the Consistency is compromised. We will discuss this aspect in detail in subsequent chapters.

MongoDB doesn't provide transactional support. Though in MongoDB the atomicity is guaranteed at the document level however if an update involves multiple documents there's no guarantee of atomicity. Similarly "isolation" is also not supported which means the data which is read by one client might be modified by another concurrent client.

# JSON based Document Store

MongoDB uses a JSON (JavaScript Object Notation) based document store to store the data. JSON /BSON (Binary JSON) provides for a schema less model which provides flexibility in terms of database design. Unlike RDBMSs' the database design is schema less and flexible and changes can be done to the schema seamlessly.

This design also makes it high performance by providing for grouping of relevant data together internally and making it easily searchable.

A JSON Document contains the actual data and is comparable to a row in SQL. However in contrast to a RDBMS row the documents have dynamic schema. Which means documents in a same collection can have different fields or structure or maybe common fields can have different type of data.

A document is a set of Key-Value pairs.

 Let's understand this with an example:

```
{  
  "Name": "ABC",  
  "Phone": ["11111111",  
            "222222"  
          ],  
  "Fax": ...  
}
```

Keys and Values come in Pairs. The Value of a key in a document can be left blank. Thus in the above example the document has three keys namely "Name", "Phone" and "Fax" and the "Fax" key has no value.

## Performance vs. Features

In order to make MongoDB high performance and fast there were certain features which may be commonly available in RBMS systems which are not available in MongoDB.

Things like Unique Key Constraints, Multi-Document Updates are not available in MongoDB.

MongoDB is a document-oriented DBMS where data is stored as documents. It does not support joins or transactions, however it provides support for secondary indexes, it enables users to query using query documents, provides support for atomic updates at per document level.

It provides replica sets which are master-slave replication with automated failover and built-in horizontal scaling via automated range-based partitioning.

# Running the Database Anywhere

One of the main design decisions was the ability to run the database from anywhere – which means it should be able to run on servers, VMs or even on the cloud using the pay-for-what-you-use service.

The language used for implementing MongoDB is C++ which enables MongoDB to achieve this goal. The 10gen site provides binaries for different OS platforms enabling MongoDB to practically run on almost any type of machines be it physical, virtual or even in cloud.

## SQL Comparison

In this part of the chapter before concluding we will briefly summarize points of how MongoDB is different from SQL.

1. MongoDB uses “document” for storing its data which can have flexible schema (documents in same collection can have different fields) enabling the users to store nested or multi-value fields such as arrays, hash etc. Whereas in RDBMS it’s a fixed schema where a column’s value should have similar data type also we cannot store arrays or nested values in the cell.
2. MongoDB doesn’t provide support for “JOIN” operations like in SQL. However it enables the user to store all relevant data together in a single document avoiding at the periphery the usage of JOINS. It has a workaround to overcome this issue. We will be discussing the same in more details in the Data Modeling consideration section.
3. MongoDB doesn’t provide support for “Transaction” in the ways of SQL. However it guarantees Atomicity at document level. Also it doesn’t guarantee “Isolation” which means a data being read by one client can have its values modified by another client concurrently. We will also look what solution is

available to overcome this issue of MongoDB is the coming sections.

# Summary

In this chapter we got to know what MongoDB is, it's history and brief details on design of the MongoDB system. In the next chapters we will understand about MongoDB's Data Model.

## MongoDB – Data Model

*“MongoDB is designed to work with documents without any need of predefined columns or data types the way relational databases are making the data model extremely flexible.”*

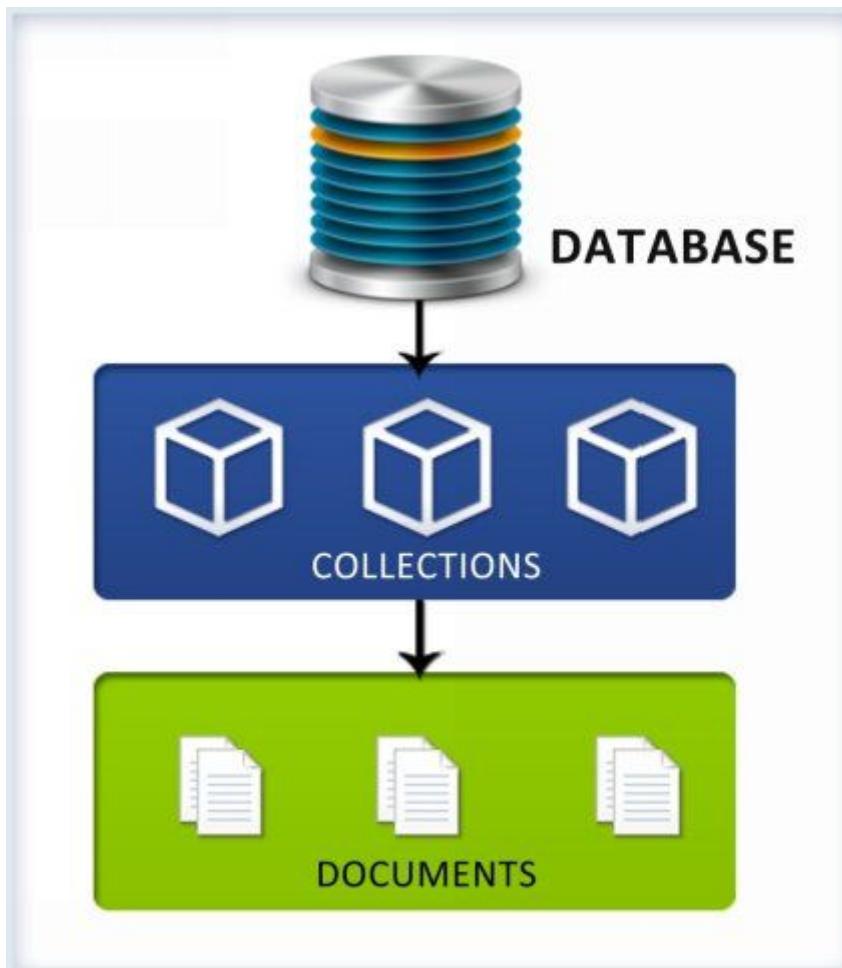
In this chapter we will introduce you to MongoDB Data Model. We will also introduce the user to what flexible schema (Polymorphic Schema) means and why it's a significant contemplation of MongoDB data model.

### The Data Model

In the previous chapter we have seen that MongoDB is a document based database which is designed to work with documents where the documents can have flexible schema which means that documents within a collection can have different (or same) set of fields. This enables the users with more flexibility when dealing with data.

In this chapter we will understand about MongoDB flexible data model and wherever required we will demonstrate the difference in the approach as compared to RDBMS.

A MongoDB deployment can have many databases. Each database is a set of collections. Collections are similar to the concept of tables in SQL however it is schema less.



***Fig 4-1: MongoDB Database Model***

Each collection can have multiple documents. Think of a document as a row in SQL but schema less.

In an RDBMS system since the table structures and the data types for each column are fixed, one can only add data of a particular data type in a column. In case of MongoDB a collection is a collection of documents where data is stored as Key-Value pairs.



*Let's understand with an example how data is stored in a document. The following document holds the Name and Phone Numbers of the Users.:*

```
{“Name”: “ABC”,  
“Phone”: [“11111111”,  
“222222”] }
```

Dynamic schema implies that documents within the same collection can have same or different set of fields or structure, and even common fields can store different type of values across documents. This implies there's no rigidness in the way data is stored in the documents of a collection.



*Let's take an example of a Region Collection:*

```
{  
“R_ID” : “REG001”,  
“Name” : “United States”  
}  
{  
“R_ID” : 1234,  
“Name” : “New York” ,  
“Country” : “United States”  
}
```

*In the above example we have two documents in the Region collection.*

*If we will observe we will find that though both the documents are part of the single collection but they have different structure with the second collection having an additional field of information which is country. In fact if we look at the “R\_ID” field it stores a STRING value in the first document whereas it's a number in the second document.*

Thus a collection's documents can have entirely different schema. It depends on the application to store documents in a particular collection together or have multiple collections. As such there is no performance difference between having multiple collections or a single collection.

## JSON AND BSON

We have seen MongoDB is a document based database. It uses Binary JSON for storing its data.

In this section we will briefly understand what are JSON and Binary-JSON (BSON).

JSON stands for JavaScript Object Notation. It's a standard used for data interchange in today's modern web (along with XML). The format is human and machine readable. It is not only a great way for exchanging data but also a nice way for storing data.

All the basic data types such as Strings, Number, Boolean Values along with Arrays, Hashes are supported by JSON.

 *Let's take an example to see how a JSON document looks like*

```
{  
  "_id" : 1,  
  "name" : { "first" : "John", "last" : "Doe" },  
  "publications" : [  
    {  
      "title" : "First Book",  
      "year" : 1989,  
      "publisher" : "publisher1"  
    },  
    { "title" : "Second Book",  
      "year" : 1999,  
      "publisher" : "publisher2"  
    }  
  ]  
}
```

```
}
```

```
]
```

```
}
```

JSON enables the users to keep all the related piece of information together at one place hence providing excellent performance. It also enables the updation to a document to be independent of each other. It is schema-less.

#### Binary JSON (BSON)

MongoDB stores the JSON document in binary encoded format. This is termed as BSON. The BSON Data model is an extended form of JSON data model.

MongoDB's this implementation of BSON document is fast, highly traversable and lightweight.

It supports embedding or arrays and objects within other arrays also enables MongoDB to reach inside the objects to build indexes and match objects against query expression both on top-level and nested BSON keys.

## The Identifier (`_id`)

We have seen till now that MongoDB stores data in documents. Documents are comprised of key-value pairs. Though a document can be compared to a row in RDBMS however unlike a row documents have flexible schema. A key which is nothing but a label can be roughly compared to the column name in RDBMS. Key is used for querying data from the documents. Hence like RDBMS primary key (used to uniquely identify each row) we need to have a key which uniquely identify each document within a collection. This is referred as `_id` in MongoDB.

If user has not explicitly specified any value for this key a unique value will be automatically generated and assigned to this key by MongoDB.

This key value is immutable and can be of any data type except arrays.

# Capped Collection

We are already well versed with Collections and Documents. Let's talk about a special type of a collection called "Capped Collection"

MongoDB has a concept of capping the collection. This means it stores the documents in the collection in inserted order and as the collection reaches its limit the documents will be removed from the collection in FIFO (First in First Out) order. This implies that the least recently inserted documents will be removed first.

The above features are good for use cases where the order of insertion is required to be maintained automatically and deletion of records after a fix size is required. Some examples of such use cases can be log files which get automatically truncated after a certain size.

 *MongoDB itself uses capped collection for maintaining its replication logs.*

Capped collection guarantees preservation of the data in insertion order, hence

1. Queries retrieving data in insertion order return results quickly and don't need an index.
2. Updates that change the document size are not allowed.

# Polymorphic Schemas

As the readers are already conversant with the schemaless nature of MongoDB data structure lets understand polymorphic schemas and the use cases.

A polymorphic schema is a schema where a collection has documents of different types or schemas.

 *A good example of this schema is a “Users” Collection, some user documents might have extra Fax number, email addresses while some might have only phone numbers but still all these documents coexist within the same “Users” collection.*

This schema is generally referred as Polymorphic schema.

In this part of the chapter, we'll explore the various reasons for using a polymorphic schema.

## Object-Oriented Programming

The Object Oriented programming enables the programmers to have classes share data and behaviors using inheritance. It also enables them to define functions in the parent class which can be overridden in the child class and hence will function differently in different context i.e. we can use the same function name to manipulate the child as well as the parent class though under the hood implementations might be different. This feature is referred as polymorphism.

Hence the requirement in this case is the ability to have a schema wherein all the related set of objects or objects within a hierarchy can fit in together and can also be retrieved identically.

 *Let's understand the above with an example, if we have an application that enables the users to upload and share different content types such as html pages, documents, images, videos etc.*

*Though many of the fields will be common across all the above mentioned content types such as Name, ID, Author, Upload Date and Time but not all fields will be identical such as in case of images we will have a Binary Field which will have the image content whereas in case of the html page we will have a large text field to hold the HTML content.*

*In this scenario the MongoDB polymorphic schema will be used wherein all the content node types will be stored in the same collection say for instance “LoadContent” wherein each document will have relevant fields only.*

```
// "Document collections" - "HTMLPage"
document
{
  _id: 1,
  title: "Hello",
  type: "HTMLpage",
  text: "<html>Hi..Welcome to my world</html>"
}

...
// Document collection also has a "Picture"
document
{
  _id: 3,
  title: "Family Photo",
  type: "JPEG",
  sizeInMB: 10, .....
}
```

*This schema not only enables the users to store related data with different structures together in a same collection, it also simplifies the querying. The same collection can be used to perform queries on common fields such as fetch all contents uploaded on a particular date and time as well as queries on specific fields such as to find out images having size greater than X MB.*

Hence the object-oriented programming is one of the use cases where having polymorphic schema makes sense.

# Schema Evolution

When we are working with databases one of the most important considerations that we need to account for is the schema evolution i.e. change in schema's impact on the running application and the design should be done in a way as to have minimal or no impact on the application i.e. no or minimal downtime, no or very minimal code changes etc.

Typically Schema evolution happens by executing a migration script which upgrades the database schema from the old version to the new one. If the database is not in production than the script can be simple drop and recreation of the database however if the database is in production environment and contains the LIVE data then the migration script will be complex as the data needs to be preserved hence the script should take this into consideration. Though in MongoDB we have an Update option which can be used to update all the documents structure within a collection if there's a new addition of field but imagine the impact of doing this if we have 1000's of documents in the collection , it will be very slow and can have negative impact on the underlying applications performance, hence one of the ways of doing this is include the new structure to the new documents being added to the collection and then gradually migrate the collection in the background while the application is still running. This is one of the many use cases, where having a polymorphic schema will be advantageous.



*For e.g. say we are working with Tickets Collection, where we have documents with ticket details as shown below*

```
// "Ticket1" document (stored in "Tickets" collection)
{
  _id: 1,
  Priority: "High",
```

```
    type: "Incident",  
    text: "Printer not working"  
} .....
```

*During some point in time the application team decides to introduce a “short description” field in the ticket document structure, so the best alternative is to introduce this new field in the new ticket documents. Within the application embed a piece of code which will handle retrieving both “old style documents (without a short description field)” as well as “new style documents (with short description field)” and then gradually the old style documents can be migrate to the new style documents. Once the migration is completed if required the code can be updated to remove the piece of code which was embedded to handle the missing field.*

## Summary

In this chapter we got to know about MongoDB Data Model. We also looked at what is an identifier and capped collection means. Finally we concluded the chapter with an understanding of how the flexible schema helps.

In the following chapter we will look at how we can get started with MongoDB. We will cover the installation and configuration of MongoDB.

# MongoDB - Installation & Configuration

*“MongoDB is a cross platform database. Let’s get started with setting up our instance of MongoDB”*

In this chapter we will go over the process for installing MongoDB on Windows as well as LINUX.

## Select Your Version

MongoDB runs on most platforms. List of all the available packages can be downloaded from the MongoDB Downloads page (<http://mongodb.org/downloads>).

The correct version for your environment will depend on your Server’s Operating System and the kind of processor. MongoDB supports both 32-bit and 64-bit architecture but it’s recommended to use 64-bit in the Production environment.

 **32-bit limitations:** Since MongoDB uses memory mapped files thus limiting the 32-bit builds to around 2 GB of data. Hence it’s recommended to use 64-bit builds for production environment for performance reasons.

The Latest MongoDB production release is 2.4.9 at the time of writing this book.



**Fig 5-1: MongoDB Downloads**

Downloads for MongoDB are available for Windows, Solaris, MAC OS X and Linux.

The MongoDB download website screen is divided in the following three sections

1. Production Release (2.4.9) – 1/10/2014
2. Previous Releases (stable)
3. Development Releases (unstable)

As can be seen the Production Release is the most stable recent version available which at time of writing of the book is 2.4.9. As a new version is released the prior stable release is moved under Previous Releases section.

The development releases as the name suggest are the versions which are still under development and hence are tagged as unstable. Hence these versions can have additional features but this may not be stable as they are still in the development phase. You can use the development versions to try out new features and provide feedback to 10gen on the features and issues faced.

# Installing MongoDB under Linux

In this section we will be installing MongoDB on a LINUX system. For the following demonstration we will be using an Ubuntu Linux distribution. We can install MongoDB either manually or can use repositories. We will walk the readers through both the options.

## Installing using Repositories

Repositories are basically online directories filled with software. Aptitude is the software that can be used to install the software's on Ubuntu. Though MongoDB might be present under the default repositories but the possibility of an out-of-date version is there hence the first step is to configure aptitude to look at a custom repository.

 “apt” and “dpkg” are Ubuntu package management tools which are used to ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys.

Issue the following command to import MongoDB public.GPG key.

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

Next create /etc/apt/sources.list.d/mongodb.list file using the following command

```
echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee /etc/apt/sources.list.d/mongodb.list
```

Finally issue the following command to reload the repository

```
sudo apt-get update
```

By the end of this step aptitude is aware of the manually added repository.

Next we need to install the software. We will be issuing the following command in the shell to install the current stable version:

```
sudo apt-get install mongodb-10gen
```

*If we wish to install an unstable version from the development releases then the following commands can be issued instead of the above command:*

```
sudo apt-get install mongodb-10gen-unstable
```

The completion of the above steps confirms the successful installation of MongoDB and that's all to it.

## Installing Manually

In this section we will see how MongoDB can be installed manually. This knowledge is specifically important in the following cases:

1. When the Linux distribution doesn't use Aptitude.
2. When the version of your interest is not available through repositories or is not part of the repository.
3. When it's required to run multiple MongoDB versions simultaneously.

The first step in manual installation is to decide on the version of MongoDB to use and then download from the site. Next the package needs to be extracted using the following command:

```
$ tar -xvf mongodb-linux-i686-2.4.9\ \ \(1\).tgz  
mongodb-linux-i686-2.4.9/README
```

```
mongodb-linux-i686-2.4.9/THIRD-PARTY-NOTICES
```

```
mongodb-linux-i686-2.4.9/GNU-AGPL-3.0
```

```
mongodb-linux-i686-2.4.9/bin/mongodump
```

```
.....
```

```
mongodb-linux-i686-2.4.9/bin/mongosniff
```

```
mongodb-linux-i686-2.4.9/bin/mongod
```

```
mongodb-linux-i686-2.4.9/bin/mongos
```

```
mongodb-linux-i686-2.4.9/bin/mongo
```

As can be seen the step extracts the package content in a new directory called mongodb-linux-i686-2.4.9 (This is located under the user's current directory).

The directory contains many subdirectories and files. The main executable files are under the sub directory bin.

This completes the MongoDB installation successfully and that's all to it.

# Installing MongoDB on Windows

Installing MongoDB on windows is a simple matter of just downloading the zip file, extract the content, create the directory folder and running the application itself.

The first step is deciding on the build that need to be downloaded.

Begin by downloading the zip file with binaries of the latest stable version. Next is extracting the archive to the root of C:\. The contents are extracted to a new directory called `mongodb-win32-x86_64-xxxx-yy-zz`; located under the C:\.

After extracting the content we will now run the Command Prompt by right clicking on the command prompt and selecting run as administrator.

Issue the following commands in the command prompt

```
C:\Users\Administrator>cd \  
C:\>move c:\mongodb-win32-* c:\mongodb
```

After successful completion of the above steps, we have a directory in C:\ with all the relevant applications in the bin folder which in this case is C:\MongoDB\Bin\. That's all there is to it.

# Start Running MongoDB

We have seen how to install MongoDB after choosing the appropriate version for our platform. It's finally time to now look at how we can start running and using MongoDB.

## Preconditions

MongoDB requires data folder to store its file which by default is C:\data\db in Windows and /data/db directory in LINUX systems.

These data directories are not created by MongoDB hence before starting MongoDB the data directory needs to be manually created and we need to ensure that proper permissions are set i.e. MongoDB has read, write and directory creation permissions.

*If the MongoDB is started before creating the folder it will throw an error message and will fail to run.*

## Start the Service

Once the directories are created and permissions are in place execute the mongod application (placed under the bin directory) to start MongoDB core database service.

In continuation to our above installation the same can be started by opening the command prompt in windows (the command prompt needs to be run as administrator) and executing the following

c:\>c:\mongodb\bin\mongod.exe

In case of Linux the mongod process is started in shell.

This will start the MongoDB Database on the localhost interface and it listens for connections from mongo shell on port 27017.

As we mentioned above the folder path need to be created before starting the database which by default is c:\data\db, an alternative path can also be provided while starting the database service using the **-dbpath** parameter.

```
C:\>C:\mongodb\bin\mongod.exe --dbpath  
C:\NewDBPath\DBContents
```

# Verifying the Installation

The relevant executable will be present under the sub directory bin. Hence the following can be checked under the bin directory in order to vet the success of the installation step.

*mongod – the core database server*

*mongo – the database shell*

*mongos – the auto-sharding process*

*mongoexport – export utility*

*mongoimport – import utility*

Apart from the above listed we have several other applications available under the bin folder.

The *mongo* application launches the mongo shell which enables the users to access the database contents and fire selective queries or executes aggregation against the data in MongoDB.

The *mongod* application as we have seen above is used to start the database service or daemon as it's called.

Multiple flags can be set while launching the applications. For e.g. as we mentioned above –dbpath can be used to specify an alternative path for the database files to be stored. In order to get the list of all available options we need to include the --help flag while launching the service.

# Using the MongoDB Shell

*mongo* shell is a part of the standard MongoDB distribution. It provides a full JavaScript environment with a complete access to the JavaScript language and all standard functions as well as a full database interface for MongoDB.

Once the database services have been started, we can fire the mongo shell and start using MongoDB. This can be done using Shell in LINUX or command prompt in windows (run as administrator).

*We need to ensure that we are referring to the exact location where the executable are say for e.g. in our case the executable are available under C:\mongodb\bin\ folder in Windows environment.*

Open the command prompt (run as administrator), type mongo.exe at the command prompt and press the Return key. This will start the mongo shell.

```
C:\>C:\mongodb\bin\mongo.exe  
MongoDB shell version: 2.4.9  
connecting to: test  
>
```

*If no parameters are specified while starting the service by default it will connect to the test database on the local host instance.*

*The database will be created automatically when connected to it which is in par with the MongoDB's feature where if an attempt is made to connect to a database which doesn't exist it will automatically create one.*

We will be covering more on working with the Mongo Shell in the next chapter.

# Securing the Deployment

We have seen how we can install and start using MongoDB using default configurations. Next we need to ensure that the data which is stored within the database is secure in all aspects.

Hence in this section we will look at how we can secure our data. We will change the configuration of the default installation to ensure that our database is more secure.

## Using Authentication and Authorization

Authentication means the users will be able to access the database only if they login using the credentials which have access on the database. This disables anonymous access to the database.

After the user is authenticated authorization can be used to ensure that the user has only the required amount of access needed for accomplishing the tasks in hand.

*In MongoDB Authentication and Authorization is supported on per-database level.*

Users exist in the context of a single logical database and are stored in the system.users collection within the database.

**system.users** - This collection stores information for authentication and authorization on that database. It stores the user's credentials for authentication and users privileges information for authorization.

MongoDB uses a role-based approach for authorization e.g. of the roles are read, readWrite, readAnyDatabase etc.

*A privilege document within the system.users collection is used for storing each user roles and credentials for users who have access to the database.*

Hence a user can have multiple roles and may have different roles on different databases.

The available roles are

read – This provides a read only access of all the collections for the specified database.

readWrite – This provides a read and write access to any collection within the specified database.

dbAdmin – This enables the users to perform administrative actions within the specified database such as index management using ensureIndex, dropIndexes, reIndex, indexStats, renaming collections, create collections etc.

userAdmin – This enables the user to perform read write operation on the system.users collection of the specified database, also enables them to modify permissions for existing users and create new users. This is effectively the SuperUser role for the specified database.

clusterAdmin – This role enables the user to grant access to administration operations which affects or present information about the whole system. clusterAdmin is applicable only on the admin database, and does not confer any access to the local or config databases.

readAnyDatabase – This role enables user to read from any database in the MongoDB environment.

readWriteAnyDatabase - This role is similar to readWrite except it is for all databases.

userAdminAnyDatabase – This role is similar as userAdmin role except it applies to all databases.

dbAdminAnyDatabase – This role is same as dbAdmin, except it applies to all databases

## ***Enabling Authentication***

Authentication is disabled by default – use `auth` for enabling authentication. While starting `mongd` use `mongod --auth`. Before enabling authentication we need to have at least one admin user.

We have seen above an admin user is a user who will be responsible for creating and managing other users, this user can create or modify any other users and can assign them any privileges.

 *It is recommended that in production deployments such users are created solely for managing users and should not be used for any other roles. This is the first user that needs to be created for a MongoDB deployment and then this user can create other users in the system.*

The admin user can be created either ways that is before enabling the authentication or after enabling the authentication.

In our example we will first create the admin user and then enable the auth settings. The below steps have been executed on the windows platform

Start the `mongod` with default settings

```
C:\>C:\mongodb\bin\mongod.exe
```

```
C:\mongodb\bin\mongod.exe --help for help and startup options
```

```
Thu Feb 06 03:23:29.473 [initandlisten] MongoDB
starting : pid=2872 port=27017
```

```
.....
```

```
Thu Feb 06 03:23:30.333 [initandlisten] waiting for
connections on port 27017
```

```
Thu Feb 06 03:23:30.333 [websvr] admin web console
waiting for connections on port 28017
```

## Create the admin user

Run another instance of command prompt by running it as an administrator and execute the mongo application.

```
C:\>C:\mongodb\bin\mongo.exe
```

*MongoDB shell version: 2.4.9*

*connecting to: test*

*>*

Switch to the admin database

 admin db is a privileged databases which the user need access on to execute certain administrative commands such as creating an admin user in this example.

```
> db = db.getSiblingDB('admin')  
admin
```

Add the user with either the userAdmin role or userAdminAnyDatabase role

```
> db.addUser({user: "AdminUser", pwd: "password",  
roles:["userAdminAnyDatabase"]  
})  
{  
  "user" : "AdminUser",  
  "pwd" : "2c14878340ab813b4e83c47b88918cdc",  
  "roles" : [  
    "userAdminAnyDatabase"  
  ],  
  "_id" : ObjectId("52f37218424122fb61569a89")  
}
```

To authenticate as this user, you must authenticate against the admin database. Restart the mongod with auth settings

```
C:\>c:\mongodb\bin\mongod.exe --auth
```

```
Thu Feb 06 18:41:36.289 [initandlisten] MongoDB  
starting : pid=3384 port=27017 dbpath=\data\db\ 64-bit  
host=ANOC9
```

```
Thu Feb 06 18:41:36.290 [initandlisten] db version v2.4.9
```

```
.....
```

```
Thu Feb 06 18:41:36.314 [initandlisten] waiting for  
connections on port 27017
```

```
Thu Feb 06 18:41:36.314 [websvr] admin web console  
waiting for connections on port 28017
```

Start the mongo console and authenticate against the admin database using the AdminUser user created above.

```
C:\>c:\mongodb\bin\mongo.exe
```

```
MongoDB shell version: 2.4.9
```

```
connecting to: test
```

```
> use admin
```

```
switched to db admin
```

```
> db.auth("AdminUser", "password")
```

**1**

>

## **Create a User and enable Authorization**

In this section we will create a user and assign a role to the newly created user. We have already authenticated using the admin user as shown below

```
C:\>c:\mongodb\bin\mongo.exe
MongoDB shell version: 2.4.9
connecting to: test
> use admin
switched to db admin
> db.auth("AdminUser", "password")
```

**1**

>

Switch to Products database and create user Alice and assign read access on the product database.

```
> use products
switched to db products
> db.addUser({user: "Alice"
... , pwd:"Moon1234"
... , roles: ["read"]
... }
... )
{
  "user" : "Alice",
  "pwd" : "c5f0f00ee1b145fb9ccc3a6ad60f01a",
  "roles" : [
    "read"
  ],
  "_id" : ObjectId("52f38d7dbd14c630732401a0")
}
```

>

We will next validate that the user has read only access on the database

```
> db  
products  
> show users  
{  
  "_id" : ObjectId("52f38d7dbd14c630732401a0"),  
  "user" : "Alice",  
  "pwd" : "c5f0f00ee1b145fb9ccc3a6ad60f01a",  
  "roles" : [  
    "read"  
  ]  
}  
>
```

Next we can connect to a new mongo console and Login using Alice to the Products database to issue read only commands.

```
C:\>c:\mongodb\bin\mongo.exe -u Alice -p Moon1234  
products  
MongoDB shell version: 2.4.9  
connecting to: products
```

# Controlling access over network

In this section we will look at the configuration options that we have used for restricting the network exposures. The below is executed on the windows platform.

```
C:\>c:\mongodb\bin\mongod.exe --bind_ip 127.0.0.1  
--port 27017 --rest
```

```
Thu Feb 06 19:06:05.995 [initandlisten] MongoDB  
starting : pid=3384 port=27017 dbpath=\data\db\ 64-bit  
host=ANOC9
```

```
Thu Feb 06 19:06:05.996 [initandlisten] db version v2.4.9
```

```
.....
```

```
Thu Feb 06 19:06:06.018 [initandlisten] waiting for  
connections on port 27017
```

```
Thu Feb 06 19:06:06.018 [websvr] admin web console  
waiting for connections on port 28017
```

We have started the server with bind\_ip. bind\_ip has one value set as 127.0.0.1 which is the localhost interface. The bind\_ip settings limits the network interface on which the program will listen for incoming connections, a comma separated list of IP addresses can be specified. For our case we have restricted the mongod to listen to only the localhost interface.

 When the mongod instance is started by default it waits for any incoming connection on port 27017. We can change this using –port.

*Changing the port does not meaningfully reduce risk or limit exposures. In order to completely secure the environment we need to allow only trusted clients to connect to the port using firewalls settings.*

Changing this port also indirectly changes the port for the HTTP status interface which by default is 28017. This port is always available on the port numbered 1000 greater than the connection port. Hence for our environment it is available on X+1000.

This web page exposes diagnostic and monitoring information that includes a variety of operational data, logs and status reports regarding the database instances i.e. it provides management level statistics which can be used for administration purpose. This page is by default read-only; to make it fully interactive we will be using the REST settings. This configuration makes the page fully interactive hence enabling the administrators in troubleshooting any performance issues when encountered. Only trusted client access should be allowed on this port using Firewalls.

It is recommended to disable the HTTP Status page as well as the REST configuration in the production environment.

## Use Firewalls

Firewalls are used to control access within a network, it can be used to allow access from a specific IP address to specific IP ports or it can be used to stop any access from any untrusted hosts. This can basically be used to create a trusted environment for our mongod instance where we can specify what all IP addresses or hosts can connect to which all ports or interfaces of the mongod.

On the windows platform we have used netsh to allow all incoming traffic to port 27017, so that any application server can connect to our mongod instance.

```
C:\>netsh advfirewall firewall add rule name="Open  
mongod port 27017" dir=in action=allow  
protocol=TCP localport=27017
```

Ok.

```
C:\>
```

The rule allows all incoming traffic on port 27017, which allows the application server to connect to the mongod instance.

## Encrypt Data

We have seen that MongoDB stores the data files in the data directory which defaults to C:\data\db in Windows and /data/db in LINUX. The files which are stored in the directory are unencrypted as Mongo doesn't provide a method to automatically encrypt these files. Any attacker who has access to the file system can view the data stored in the files. Hence it's the application responsibility to ensure that the sensitive information's are encrypted before it's written to the database.

Additionally operating system level mechanisms such as file system level encryption and permissions should be implemented in order to prevent unauthorized access to the files.

In order to encrypt and secure sensitive data MongoDB has partnership with Gazzang. Gazzang provides solutions for encrypting mongoDB data and making it more secure. More information on Gazzang is available at <http://www.gazzang.com>

## Encrypt communication

Many a times it is required to ensure that the communication between the mongod and the client (mongo shell for instance) is encrypted.

Hence in this setup we will see how we can add one more level of security to our above installation by configuring SSL, so that the communication between the mongod and mongo shell (client) happens using SSL certificate and key.

 *It is recommended to use SSL for communication between the server and the client.*

*SSL is not supported in the default distribution of MongoDB. In order to use SSL you need to either build MongoDB locally passing the “--ssl” option or use MongoDB Enterprise.*

The below commands are executed on an Ubuntu system and assumes that the MongoDB installed is the build that includes SSL support with SSL support at the client driver level too.

The first step is to generate the .pem file which will contain the public key certificate and the private key. The following command generates a self-signed certificate and private key.

```
cd /etc/ssl/
```

```
sudo openssl req -new -x509 -days 365 -nodes -out  
mongodb-cert.crt -keyout mongodb-cert.key
```

Next the certificate and private key is concatenated to a .pem file

```
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

We will now include the following run-time options while starting up the mongod.

```
mongod --sslOnNormalPorts --sslPEMKeyFile  
/etc/ssl/mongodb.pem
```

We will next see how we can connect to the mongod running with SSL using mongo shell.

Start the mongo shell using –ssl and –sslPemKeyFile (this specifies the signed certificate key file) options.

```
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

*If we need to connect to a mongod instance which requires only a SSL encryption mode then we need to start the mongo shell with --ssl, as shown in the following:*

```
mongo --ssl
```

# Summary

In this chapter we looked at how to install MongoDB on a windows and LINUX platform. We also looked at few important configurations which are necessary to ensure a secure and safe usage of the database. In the following chapter we will look at how we can get started with MongoDB Shell.

## Using Mongo Shell

*“[mongo shell](#) is a part of the standard MongoDB distribution, It provides a full JavaScript environment with a complete access to the JavaScript language and all standard functions as well as a full database interface for MongoDB.”*

In this chapter we will cover how to get started with a mongo shell. Before we delve into creating applications which interact with the database it is important to understand how MongoDB shell works.

There's no better way than to get started with the MongoDB shell which comes as part of the standard MongoDB distribution, it not only provides a full JavaScript environment with all standard functions in it but it also provides a full database interface for MongoDB.

The MongoDB shell introduction has been divided into three parts for making it easier for the readers to grasp and practice the concepts: The first part covers the basic features of the database which covers the basic CRUD operators. The Next part covers advanced querying finally the chapter is concluded with the understanding of the two ways of storing and retrieving data using Embedding and Referencing.

# Part I: Basic Querying

In this section we will briefly understand about the common database operations i.e. CRUD in MongoDB where CRUD stands for Create, Read, Update and Delete.

Using basic examples and exercises we will understand how these operations are performed in MongoDB. In this process we will understand how queries are executed in MongoDB.

Instead of using a standardized query language such as SQL MongoDB uses its own JSON-like query language to retrieve information from the data stored.

After successful installation of MongoDB we have a directory in C:\ with all the relevant applications in the bin folder which in this case is C:\MongoDB\Bin\

The MongoDB shell can be started by executing the mongo executable.

*The Mongod by default listens for any incoming connection on port 27017 of the localhost interface.*

The first step is always to start the database server. Open the command prompt (by running it as administrator) and issue the command CD \. Next run the command C:\MongoDB\bin\mongod.exe (if the installation is in some other folder then accordingly the path will change, for the examples in this chapter the installation is in C:\MongoDB folder). This will start the database server.

C:\>`c:\mongodb\bin\mongod.exe`

`c:\mongodb\bin\mongod.exe --help for help and startup options`

`Fri Feb 07 12:42:40.732 [initandlisten] MongoDB starting : pid=7112 port=27017`

.....

*Fri Feb 07 12:42:40.919 [initandlisten] waiting for connections on port 27017*

*Fri Feb 07 12:42:40.919 [websvr] admin web console waiting for connections on port 28017*

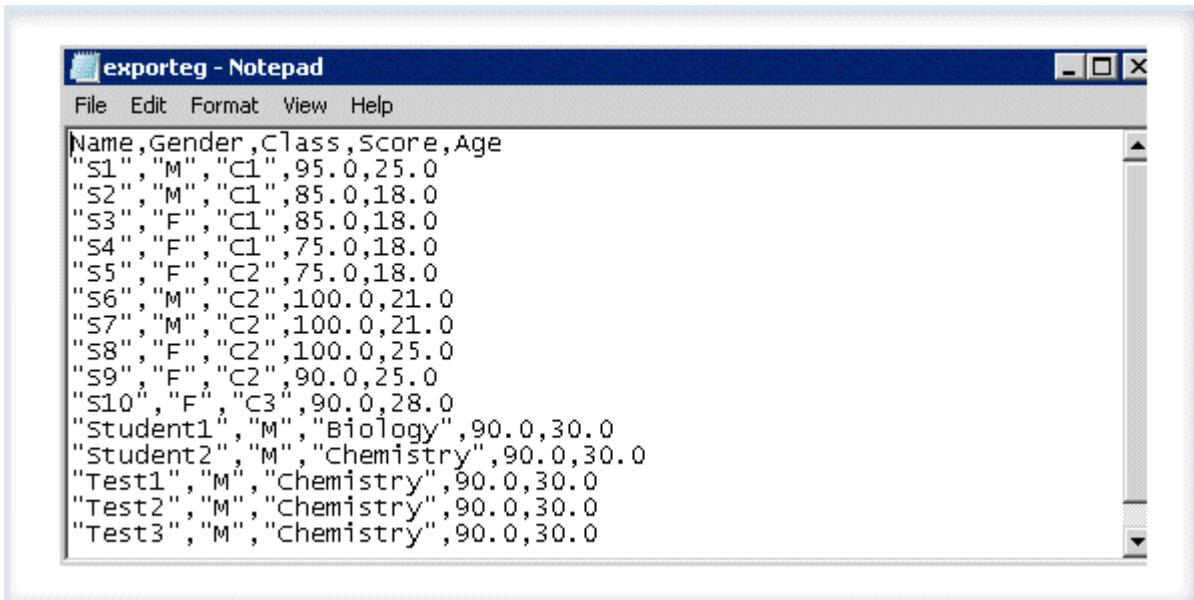
Since the database server is started now the user can start issuing command to the server using the mongo shell.

Before we look at the mongo shell we will briefly look at how we can use the import/export tool to import/export data in and out from the MongoDB database.

We will create a CSV file to hold records of students with the following structure:

*Name, Gender, Class, Score, Age.*

Sample data of the CSV is as shown below:



The screenshot shows a Microsoft Notepad window titled "exporteg - Notepad". The window contains a CSV file with the following data:

	Name	Gender	Class	Score	Age
1	"S1"	"M"	"C1"	95.0	25.0
2	"S2"	"M"	"C1"	85.0	18.0
3	"S3"	"F"	"C1"	85.0	18.0
4	"S4"	"F"	"C1"	75.0	18.0
5	"S5"	"F"	"C2"	75.0	18.0
6	"S6"	"M"	"C2"	100.0	21.0
7	"S7"	"M"	"C2"	100.0	21.0
8	"S8"	"F"	"C2"	100.0	25.0
9	"S9"	"F"	"C2"	90.0	25.0
10	"S10"	"F"	"C3"	90.0	28.0
11	"Student1"	"M"	"Biology"	90.0	30.0
12	"Student2"	"M"	"Chemistry"	90.0	30.0
13	"Test1"	"M"	"Chemistry"	90.0	30.0
14	"Test2"	"M"	"Chemistry"	90.0	30.0
15	"Test3"	"M"	"Chemistry"	90.0	30.0

**Fig 6-1: Sample CSV file**

We will next import the data to the MongoDB database to a new collection in order to look at how the import tool works.

Open the command prompt by *running it as an administrator*. The following command is used to get help on the *import* command.

```
C:\>c:\mongodb\bin\mongoimport.exe --help
```

*Import CSV, TSV or JSON data into MongoDB.*

*When importing JSON documents, each document must be a separate line of the input file.*

*Example:*

```
mongoimport --host myhost --db my_cms --collection  
docs < mydocfile.json
```

....

```
C:\>
```

Let's issue the below command to import the data from the file "exporteg.csv" to a new collection "importeg" in the MyDB database.

```
C:\>c:\mongodb\bin\mongoimport.exe --host  
localhost --db mydb --collection importeg --type csv  
--file c:\exporteg.csv --headerline  
connected to: localhost  
Fri Feb 07 12:49:33.642 check 9 16  
Fri Feb 07 12:49:33.643 imported 15 objects
```

In order to validate whether the collection is created and the data is imported, we will connect using mongo shell to the database which is *localhost* in this case and will issue commands to validate whether the collection exists or not. The commands used in the code snippet are explained in more details later in the section.

```
C:\>c:\mongodb\bin\mongo.exe  
MongoDB shell version: 2.4.9  
connecting to: test  
> use mydb  
switched to db mydb  
> show collections  
MR_ClassAvg  
importeg  
mapreducecount  
mapreducecount1  
students
```

```

system.indexes
> db.importeg.find()
{ "_id" : ObjectId("51beef99dc7601ac46127913"), "Name" : "S1", "Gender" : "M", "Class" : "C1", "Score" : 95, "Age" : 25 }

.....
{ "_id" : ObjectId("51beef99dc7601ac46127921"), "Name" : "Test3", "Gender" : "M", "Class" : "Chemistry", "Score" : 90, "Age" : 30 }
>

```

Just in brief what we are doing here is

1. Connect to the mongo shell
2. Switch to our database which is “MyDB” in this case
3. Checked for the collections that exist in the MyDB database using “*show collections*”
4. Checked the count of the collection which we imported using the import tool.
5. Finally executed *find()* command to check for the data in the new collection.

Now we will start with usage of mongo shell. To start the mongo shell, run command prompt as administrator and issue command C:\MongoDB\bin\mongo.exe (the path will vary based on the installation folder, in this example the folder is C:\MongoDB\) and press enter. This by default connects to the *localhost* database server which is listening on port 27017.

*Use –port option and –host options to connect to a server on a different port or interface.*

```
C:\>c:\mongodb\bin\mongo.exe
MongoDB shell version: 2.4.9
connecting to: test
```

>

As we can see in the above screen by default the database **test** is used for context.

At any point of time executing *db* command will show the current database the shell is connected to.

> **db**

*test*

>

In order to display all the database names the user can run *show dbs* command: executing the command will list down all the databases for the connected server.

> **show dbs**

At any point help can be accessed using the *help()* command.

> **help**

*db.help() help on db methods  
db.mycoll.help() help on collection methods  
sh.help() sharding helpers  
rs.help() replica set helpers  
help admin administrative help  
help connect connecting to a db help  
help keys key shortcuts  
help misc misc things to know  
help mr mapreduce  
show dbs show database names  
show collections show collections in current database  
show users show users in current database*

.....  
*exit quit the mongo shell*

As shown above if we need help on any of the methods of db or collection we can use *db.help()* or *db.<CollectionName>.help()*. For example if we need help on the *db* command we will execute *db.help()*.

> **db.help()**

*DB methods:*

*db.addUser(userDocument)*

...

*db.shutdownServer()*

*db.stats()*

*db.version() current version of the server*

>

Till now we have been using the default *test* db. The command *use <newdbname>* can be used to switch to the new database.

> **use mydb**

*switched to db mydb*

>

Before we start our exploration lets first briefly look at the MongoDB terminology and concepts corresponding to the SQL terminology and concepts. This is summarized in the table below

**Table 6-1: SQL and MongoDB terminology**

SQL	MongoDB
<b>Database</b>	Database
<b>Table</b>	Collection
<b>Row</b>	Document
<b>Column</b>	Field
<b>Index</b>	Index
<b>Table joins</b>	Embedded documents and Linking
<b>Primary Key – A column or a group of column can be specified as Primary Key</b>	Primary Key – Automatically set to <code>_id</code> field

Let's start our exploration of understanding the querying in MongoDB. Switch to *MYDBPOC* database.

```
> use mydbpoc
switched to db mydbpoc
>
```

This switches the context from *test* to *MYDBPOC*. The same can be confirmed using the `db` command.

```
> db
mydbpoc
>
```

Though the context is switched to *MYDBPOC* but the database name will not appear if the `show dbs` command is issued because MongoDB doesn't create a database until data is inserted into the

database. This is in par with the MongoDB's dynamic approach to data facilitating dynamic namespace allocation and a simplified and accelerated development process.

 *If we issue show dbs command at this point it will not list MYDBPOC database in the list of databases, as the database is not created until data is inserted into the database.*

*The following example assumes a polymorphic collection named users which contains the documents of the following prototypes:*

```
{  
  _id: ObjectId(),  
  FName: "First Name",  
  LName: "Last Name",  
  Age: 30,  
  Gender: "M",  
  Country: "Country"  
}  
and  
{  
  _id: ObjectId(),  
  Name: "Full Name",  
  Age: 30,  
  Gender: "M",  
  Country: "Country"  
}  
and  
{  
  _id: ObjectId(), Name: "Full Name", Age: 30 }
```

## Create and Insert

We will now look at how database and collections are created. The documents will be specified in JSON.

First by issuing `db` command we will confirm that the context is `mydbpoc` database.

```
> db  
mydbpoc  
>
```

We will first see how we can create documents.

The first document complies with the first prototype whereas the second document complies with the second prototype. We have created two variables `user1` and `user2`.

```
> user1 = {FName: "Test", LName: "User", Age:30,  
Gender: "M", Country: "US"}  
{  
  "FName" : "Test",  
  "LName" : "User",  
  "Age" : 30,  
  "Gender" : "M",  
  "Country" : "US"  
}  
> user2 = {Name: "Test User", Age:45, Gender: "F",  
Country: "US"}  
{ "Name" : "Test User", "Age" : 45, "Gender" : "F",  
"Country" : "US" }  
>
```

We will next add both these documents i.e. `user1` and `user2` to the `users` collection using the following sequence of operations.

```
> db.users.insert(user1)  
> db.users.insert(user2)  
>
```

The above operation will not only insert the two documents to the `users` collection but it will also create the collection as well as the

database the same can be verified using the *show collections* and *show dbs* command.

*show collections* will display the list of collection in the current database and *show dbs* as mentioned above will display the list of databases.

```
> show dbs
admin 0.203125GB
local 0.078125GB
mydb 0.203125GB
mydb1 (empty)
mydbnew 0.203125GB
mydbpoc 0.203125GB
products 0.203125GB
> show collections
system.indexes
users
>
```

As seen in the collections screenshot along with the collection *users*, *system.indexes* collection is also getting displayed. This *system.indexes* collection is created by default when the database is created. This manages the information of all the indexes of all collections within the database.

Executing the command *db.users.find()* will display the documents in the *users* collection.

```
> db.users.find()
{ "_id" : ObjectId("52f48cf474f8fdcfcae84f79"), "FName" :
"Test", "LName" : "User", "Age" : 30, "Gender" : "M",
"Country" : "US" }
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" :
"Test User", "Age" : 45
, "Gender" : "F", "Country" : "US" }
>
```

We can see the two documents we created being displayed. In addition to the fields we added to the document there's an additional `_id` field which is being generated for all the documents.

 *All documents must have a unique `_id` field. If not explicitly specified by the user the same will be auto assigned as a unique Object ID by MongoDB as happened in our example above. We didn't explicitly insert an `_id` field but when we use `find()` command to display the documents we can see an `_id` field associated with each document.*

*The reason behind this is by default an index is created on the `_id` field which can be validated by issuing `find` command on the `system.indexes` collection.*

```
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydbpoc.users",
  "name" : "_id_" }
>
```

New indexes can be added or removed from the collection using `ensureIndex()` and `dropIndex()` command. This we will covering later in this chapter. By default an index is created on the `_id` field of all the collections. *This default index cannot be dropped.*

## Explicit Create Collection

In the above example the first insert operation implicitly created the collection. However the user can also explicitly create a collection before executing the insert statement.

```
db.createCollection("users")
```

## Insert documents using Loop

In the above example we created two document variables and inserted them to a collection one by one. Documents can also be added to the collection using `for` Loop. In the next example we will insert users using `for` loop.

```
> for(var i=1; i<=20; i++) db.users.insert({ "Name" :
  "Test User" + i, "Age": 10+i, "Gender" : "F",
  "Country" : "India" })
```

>

In order to verify that the save is successful we will run the *find* command on the collection

> db.users.find()

```
{ "_id" : ObjectId("52f48cf474f8fdcfcae84f79"), "FName" :  
"Test", "LName" : "User", "Age" : 30, "Gender" : "M",  
"Country" : "US" }
```

```
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" :  
"Test User", "Age" : 45  
, "Gender" : "F", "Country" : "US" }
```

```
.....  
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8c"), "Name" :  
"Test User18", "Age" :  
28, "Gender" : "F", "Country" : "India" }
```

Type "it" for more

>

Users appear in the collection. Before we move any further let's understand what the below statement means "Type "it" for more".

The *find* command returns a cursor to the result set. Instead of displaying all documents(which can be thousands or millions of results) in one go on the screen the cursor displays first 20 documents and wait for the request to iterate (*it*) to display the next 20 and so on till all the resultset is displayed.

The resulting cursor can also be assigned to a variable and then programmatically it can be iterated over using a *while* loop. The cursor object can also be manipulated as an array.

In our case if we type "it" and press enter then the following screen will appear.

> it

```
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8d"), "Name" :  
"Test User19", "Age" :  
29, "Gender" : "F", "Country" : "India" }
```

```
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8e"), "Name" :  
"Test User20", "Age" :  
30, "Gender" : "F", "Country" : "India" }  
>
```

Since only two documents were remaining it has displayed the two documents.

## Insert with explicitly specifying `_id`

In the previous examples of Insert the `_id` field was not specified hence was implicitly added. In the following example we will see how we can explicitly specify `_id` field while inserting the documents within a collection. While explicitly specifying the `_id` field we have to keep in mind the uniqueness of the field otherwise the insert will fail.

The following command explicitly specify the `_id` field

```
> db.users.insert({"_id":10, "Name": "explicit id"})
```

The insert operation creates the following document in the `users` collection:

```
{ "_id" : 10, "Name" : "explicit id" }
```

This can be confirmed by issuing the following command:

```
>db.users.find()
```

## Update

We now know how to create collections and insert documents into the same. Before we move ahead and start retrieving the documents using queries we must understand how we can modify the schema structure.

When working in a real world application we always come across schema evolution where we might end up adding or removing fields from the documents. We will next see how we can perform these alterations in the MongoDB database.

We have seen that there's no structure enforcement at the collection level hence there's no update of structure at the collection level. However update () operations can be used at the document level to update an existing document or set of documents within a collection.

The *update()* method by default updates a single document but by using multi option this can be used to update all documents that match the selection criteria.

Let's begin by updating the values of existing columns. The \$set operator will be used for updating the records. The following command updates the country to UK for all Female users.

```
> db.users.update({"Gender":"F"}, {$set: {"Country":"UK"}})
```

To check whether the update has happened we will issue a find command to check all the female users.

```
> db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" :
"Test User", "Age" : 45
, "Gender" : "F", "Country" : "UK" }
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f7b"), "Name" :
"Test User1", "Age" : 11, "Gender" : "F", "Country" :
"India" }
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f7c"), "Name" :
"Test User2", "Age" : 12, "Gender" : "F", "Country" :
"India" }
```

.....  
Type "it" for more

```
>
```

If we check the output we will see only the first document record is updated which is the default behavior of update as no multi option was specified.

Now let's change the update command and include the multi option.

```
>db.users.update({"Gender":"F"},{$set: {"Country":"UK"}}, {multi:true})
```

>

Now again we will issue the find command to check whether the Country is updated for all the female employees or not. Issuing the find command will return the following output.

```
> db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" :
"Test User", "Age" : 45, "Gender" : "F", "Country" : "UK" }
```

.....

Type "it" for more

>

As seen the country is updated to UK. Hence if we need to update all documents matching the criteria than we need to set the option multi to true otherwise it will update only the first document matching the criteria.

We will next look at how we can add new fields to the documents. In order to add fields to the document we will be using update() command with \$set operator and multi option only. If we use a field name with \$set which is non-existent then the field will be added to the documents. The following command will add the field "company" to all the documents

```
> db.users.update({},{$set:
{"Company":"TestComp"}},{multi:true})
>
```

Issuing find command against the user's collection, we will find the new field added to all documents.

```
> db.users.find()
{ "Age" : 30, "Company" : "TestComp", "Country" : "US",
"FName" : "Test", "Gender" : "M", "LName" : "User", "_id"
: ObjectId("52f48cf474f8fdcfcae84f79") }
{ "Age" : 45, "Company" : "TestComp", "Country" : "UK",
"Gender" : "F", "Name" : "Test User", "_id" :
ObjectId("52f48cfb74f8fdcfcae84f7a") }
{ "Age" : 11, "Company" : "TestComp", "Country" : "UK",
"Gender" : "F", .....}
```

*Type "it" for more*

>

Hence if we execute update () command with fields existing in the document, it will update the fields value, however if the field is not present in the document then the field will be added to the documents.

We will next see how we can use the same update() command with \$unset operator to remove fields from the documents.

The following command will remove the field company from all the documents.

```
> db.users.update({}, {$unset: {"Company": ""}},  
{multi:true})  
>
```

The same can be checked by issuing find() command against the Users collection.

```
> db.users.find()  
{ "Age" : 30, "Country" : "US", "FName" : "Test",  
"Gender" : "M", "LName" : "User", "_id" :  
ObjectId("52f48cf474f8fdcfcae84f79") }
```

*Type "it" for more*

## Delete

Having covered how we can insert documents in a collection and how we can change document structure by adding/removing fields of the document, we will see how we can delete data from the database. To delete documents in a collection we use the remove () method. If we specify a selection criteria only the documents meeting the criteria will be deleted, if no criteria is specified then all the documents will be deleted.

The following command will delete the documents where the Gender = 'M'.

```
> db.users.remove({"Gender": "M"})
```

>

The same can be verified by issuing the find() command on Users

> `db.users.find({"Gender":"M"})`

>

No documents are returned.

Next command will delete all documents.

> `db.users.remove()`

> `db.users.find()`

>

As we can see no documents are returned.

Finally if we want to drop the collection the following command will drop the collection.

> `db.users.drop()`

*true*

>

In order to validate whether the collection is dropped or not we will issue the show collections command.

> `show collections`

*system.indexes*

>

As we can see above the collection name is not displayed hence confirming that the collection is removed from the database.

Having covered the basic Create, Update and Delete operation, next in this section we will look at how we perform Read operation.

## Read

In this part of the chapter we will look at various examples illustrating the querying functionality available as part of MongoDB which enables the users to read the stored data from within the database.

In order to start with basic querying we will again first create the users collection and insert data following the insert commands.

```

> user1 = {FName: "Test", LName: "User", Age:30,
Gender: "M", Country: "US"}
{
  "FName" : "Test",
  "LName" : "User",
  "Age" : 30,
  "Gender" : "M",
  "Country" : "US"
}
> user2 = {Name: "Test User", Age:45, Gender: "F",
Country: "US"}
{ "Name" : "Test User", "Age" : 45, "Gender" : "F",
"Country" : "US" }
> db.users.insert(user1)
> db.users.insert(user2)
> for(var i=1; i<=20; i++) db.users.insert({"Name" :
"Test User" + i, "Age": 10+i, "Gender" : "F",
"Country" : "India"})
>

```

We will next start with basic querying.

The `find()` command is used for retrieving data from the database. The basic `find()` command we have been using till now in this chapter. Firing a `find()` command returns all the documents within the collection.

```

> db.users.find()
{ "_id" : ObjectId("52f4a823958073ea07e15070"),
"FName" : "Test", "LName" : "User", "Age" : 30, "Gender"
: "M", "Country" : "US" }
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name"
: "Test User", "Age" : 45, "Gender" : "F", "Country" : "US"
}
.....
{ "_id" : ObjectId("52f4a83f958073ea07e15083"), "Name"
: "Test User18", "Age" : 28, "Gender" : "F", "Country" :

```

```
"India" }  
Type "it" for more  
>
```

## ***Query Documents***

MongoDB provides a rich query system to filter the documents in a collection.

In order to achieve the same “query documents” can be passed as parameter to the find method.

A Query Document is specified within open “{“ and closed “}” curly braces. A query document is matched against all the documents in the collection before returning the result set

Using find() command without any query document or an empty query document i.e. find({}) returns all the documents within the collection.

A Query document can contain selectors and projectors. A selector is like a where condition in SQL or a filter which is used to filter out the results. A projector is like the select condition or the selection list which is used to display the data fields.

## **Selector**

We will now see how to use the selector.

The following command will return all the female users.

```
> db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name"
: "Test User", "Age" : 45, "Gender" : "F", "Country" : "US"
}
```

```
.....
{ "_id" : ObjectId("52f4a83f958073ea07e15084"), "Name"
: "Test User19", "Age" : 29, "Gender" : "F", "Country" :
"India" }
```

Type "it" for more

```
>
```

Let's step it up a notch. MongoDB also supports operators where we can merge different conditions together in order to refine our search on the basis of our requirements. Let's refine the above query, let's look for Female users from Country India. The below command will return the same.

```
>           db.users.find({"Gender":"F",           $or:
[{"Country":"India"}]})
```

```
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name"
: "Test User1", "Age" : 11, "Gender" : "F", "Country" :
"India" }
```

```
.....
{ "_id" : ObjectId("52f4a83f958073ea07e15085"), "Name"
: "Test User20", "Age" : 30, "Gender" : "F", "Country" :
"India" }
```

```
>
```

Next if we want to find all Female users who belong to either India or US then we will execute the following command:

```
>db.users.find({"Gender":"F",$or:
[{"Country":"India"}, {"Country":"US"}]})
```

```
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45, "Gender" : "F", "Country" : "US" }
```

```
.....
```

```
{ "_id" : ObjectId("52f4a83f958073ea07e15084"), "Name" : "Test User19", "Age" : 29, "Gender" : "F", "Country" : "India" }
```

Type "it" for more

>

Hence executing a *find()* command returns the list of users as an output satisfying the conditions we have specified as selector in the query document. Say if the requirement is to just know the count of users, or perform some other aggregation on the result set, rather than simply printing the list, in such cases aggregate functions need to be used. We will be looking into the aggregate functions in more details in coming examples.

As of now we will just look at how to know the count of records. Say for e.g. in our above example instead of displaying the documents we want to find out count of female users who stay in either "India" or "US", in that case we will execute the following command.

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"}, {"Country":"US"}]}).count()
```

## 21

>

If we want to find out count of users irrespective of any selectors then we need to execute the following command.

> `db.users.find().count()`

**22**

>

## ***Projector***

We have seen how we can use selector to filter out the documents within the collection. In the above example the `find()` command returns all fields of the documents matching the selector. Let's add projector to the Query document wherein in addition to the selector we will also mention specific details or fields that need to be displayed for e.g. let's suppose we want to display the first name and age of all female employees. In this case along with the selector projector is also used, i.e. instead of displaying the complete document we will be displaying few fields from the document. We will execute the following command to return the desired result set.

```
> db.users.find({"Gender":"F"}, {"Name":1,"Age":1})  
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name"  
: "Test User", "Age" : 45 }
```

.....

Type "it" for more

>

## ***sort()***

In MongoDB the sort order is specified as follows: 1 for Ascending and -1 for Descending sort. If in the above example we want to sort the records by ascending order of age then we will execute the following command.

```
>db.users.find({"Gender":"F"},  
{"Name":1,"Age":1}).sort({"Age":1})  
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name"  
: "Test User1", "Age" : 11 }  
{ "_id" : ObjectId("52f4a83f958073ea07e15073"), "Name"  
: "Test User2", "Age" : 12 }
```

```
{ "_id" : ObjectId("52f4a83f958073ea07e15074"), "Name" : "Test User3", "Age" : 13 }
```

```
.....  
{ "_id" : ObjectId("52f4a83f958073ea07e15085"), "Name" : "Test User20", "Age" : 30 }
```

Type "it" for more

If we want to display the records by Descending Order in Name and Ascending Order in Age then we will be executing the following command:

```
>db.users.find({"Gender":"F"},  
{"Name":1,"Age":1}).sort({"Name":-1,"Age":1})  
{ "_id" : ObjectId("52f4a83f958073ea07e1507a"), "Name" : "Test User9", "Age" : 19 }  
{ "_id" : ObjectId("52f4a83f958073ea07e15079"), "Name" : "Test User8", "Age" : 18 }  
{ "_id" : ObjectId("52f4a83f958073ea07e15078"), "Name" : "Test User7", "Age" : 17 }
```

```
.....  
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1", "Age" : 11 }
```

Type "it" for more

>

## ***limit()***

We will now look at how we can limit the records. For e.g. in case of huge collections with thousands of documents if we want to return only 5 matching documents then in that case *limit* command is used which enables us to do exactly the same.

Let's say in our previous query of Female users who belongs to either country India or US, we want to limit the result set and return only two users. Then the following command needs to be executed

```
>db.users.find({"Gender":"F"},{$or:  
[{"Country":"India"}, {"Country":"US"}]}).limit(2)
```

```
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User", "Age" : 45, "Gender" : "F", "Country" : "US" }  
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1", "Age" : 11, "Gender" : "F", "Country" : "India" }
```

## ***skip()***

If the requirement is to skip the first two records and return the 3<sup>rd</sup> and 4<sup>th</sup> user then *skip* command is used. The following command needs to be executed

```
>db.users.find({"Gender":"F"},{$or:  
[{"Country":"India"},  
 {"Country":"US"}]}).limit(2).skip(2)  
{ "_id" : ObjectId("52f4a83f958073ea07e15073"), "Name" : "Test User2", "Age" : 12, "Gender" : "F", "Country" : "India" }  
{ "_id" : ObjectId("52f4a83f958073ea07e15074"), "Name" : "Test User3", "Age" : 13, "Gender" : "F", "Country" : "India" }  
>
```

## ***findOne()***

Similar to *find()* we have a *findOne()* command which returns a single document from the collection. The *findOne()* methods takes the same parameters as *find()*, but rather than returning a cursor it returns a single document. Say for e.g. we want to return one female user who stays in either India or US. This can be achieved using the following command.

```
> db.users.findOne({"Gender":"F"},  
{"Name":1,"Age":1})  
{  
  "_id" : ObjectId("52f4a826958073ea07e15071"),  
  "Name" : "Test User",  
  "Age" : 45  
}  
>
```

Similarly if we want to return the first record irrespective of any selector in that case also we can use *findOne()* and that will return the first document in the collection.

```
> db.users.findOne()  
{  
  "_id" : ObjectId("52f4a823958073ea07e15070"),  
  "FName" : "Test",  
  "LName" : "User",  
  "Age" : 30,  
  "Gender" : "M",  
  "Country" : "US"}
```

## Using Cursor

When the *find()* method is used, MongoDB returns the results of the query as a cursor object. The mongo shell then iterates over the cursor to display the results. The maximum number of documents that are iterated and displayed on the screen is 20. When a user executes the *it* command, the shell iterates over the next set of 20 records. Thus all records are not iterated for display in one go.

MongoDB enables the users to work with the Cursor object of the find method. In the next example we will see how the user can store the cursor object in a variable and manipulate it using a *while* loop.

Say for our example we want to return all the users in the Country: US. We will declare a variable and assign the resulting cursor object

to that variable. We then print the full result set using the while loop to iterate over the variable.

The code snippet is as below:

```
> var c = db.users.find({"Country":"US"})
> while(c.hasNext()) printjson(c.next())
{
  "_id" : ObjectId("52f4a823958073ea07e15070"),
  "FName" : "Test",
  "LName" : "User",
  "Age" : 30,
  "Gender" : "M",
  "Country" : "US"
}
{
  "_id" : ObjectId("52f4a826958073ea07e15071"),
  "Name" : "Test User",
  "Age" : 45,
  "Gender" : "F",
  "Country" : "US"
}
>
```

The `hasNext()` function returns true if the cursor has documents. The `next()` method returns the next document. The `printjson()` method renders the document in a JSON-like format.

The variable to which the cursor object is assigned can also be manipulated as an array. Say for example instead of looping through the variable we want to display the document at array index 1 then in that case we can run the following command:

```
> var c = db.users.find({"Country":"US"})
> printjson(c[1])
{
  "_id" : ObjectId("52f4a826958073ea07e15071"),
  "Name" : "Test User",
```

```
.... "Gender" : "F",
      "Country" : "US"}
```

```
>
```

We have seen how we can do basic querying, sorting, limiting etc. We also saw how we can manipulate the result set using a while loop or as an array. In the next section we will look at indexes.

 *Important Note: Notice that in the query, the value portion need to be determined before the query is made (in other words, it cannot be based on other attributes of the document). For example, let's say if we have a collection of "Persons", it is not possible to express a query that returns a person whose weight is larger than 10 times of their height.*

## **explain()**

The `explain()` function can be used to see what steps the MongoDB database is running while executing a query. The following command will display the steps executed while filtering on the `username` field.

```
> db.users.find({"Name":"Test User"}).explain()
{
```

```
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 22,
  "nscanned" : 22,
  "nscannedObjectsAllPlans" : 22,
  "nscannedAllPlans" : 22,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
```

```
},  
"server" : "ANOC9:27017"}  
  
}
```

# Using Indexes

Indexes provide high performance read operations for frequently used queries. By default whenever a collection is created and documents are added to it an index is created on the `_id` field of the document. In this section we will look at how different types of indexes can be created. Let's begin by inserting 1 million documents using `for` loop in a new collection say `testindx`.

```
>for(i=0;i<1000000;i++)
{db.testindx.insert({"Name":"user"+i,"Age":Math.floor(Math.random()*120)})}
```

Next we will run `find()` command to fetch a `Name` with value as `user101`. We will run the `explain()` command to check what are the steps MongoDB is executing in order to return the result set.

```
> db.testindx.find({"Name":"user101"}).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1000000,
  "nscanned" : 1000000,
  "nscannedObjectsAllPlans" : 1000000,
  "nscannedAllPlans" : 1000000,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 464,
  "indexBounds" : {

  },
  "server" : "ANOC9:27017"
}
```

```
}
```

```
>
```

As we can see in the above example the database has scanned the entire table. This has significant performance impact and is happening because there are no indexes.

## **Single Key Index**

We will next create an index on *Name* field of the document. *ensureIndex()* is used to create index.

```
> db.testindx.ensureIndex({"Name":1})
```

The index creation will take few minutes depending on the server and the collection size. Let us run the same query that we ran earlier with *explain()* to check what are the steps the database is executing post index creation. Check the “*n*”, “*nscanned*” and “*millis*” fields in the output.

```
> db.testindx.find({"Name":"user101"}).explain()
```

```
{
```

```
  "cursor" : "BtreeCursor Name_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0, "nChunkSkips" : 0, "millis" : 2,
  "indexBounds" : { "Name" : [
    [
      "user101", "user101"
    ]
  ]}
```

```
"server" : "ANOC9:27017"  
}  
>
```

As you can see in the results, there is no table scan. The index creation makes a significant difference in the query execution time.

The index we created above is a single-key index.

## Compound Index

While creating an index we should keep in mind that the index covers most of our queries. Say for e.g. if we query sometimes on only the *Name* field and at times we query on both the *Name* and the *age* field then in that scenario creating a compound index will be more efficient than a single-key index, as the compound index will be used for both queries.

The following command creates a compound index on *Name* and *Age* fields of the *testindx* Collection.

```
> db.testindx.ensureIndex({"Name":1, "Age": 1})
```

Compound indexes help MongoDB execute queries with multiple clauses more efficiently.

While creating a compound index it is also very important to keep in mind that the fields which will be used for exact matches (for e.g. “*Name*”: “S1”) comes first followed by fields which are used in ranges (e.g. “*Age*”: {"\$gt":20}).

Hence the above index will be beneficial for the below query.

```
>db.testindx.find({"Name": "user5", "Age":  
{$gt:25}}).explain()  
{  
  "cursor" : "BasicCursor",  
  "isMultiKey" : false,  
  "n" : 0,  
  "nscannedObjects" : 22,  
  "nscanned" : 22,  
  "nscannedObjectsAllPlans" : 22,
```

```

    "nscannedAllPlans" : 22,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "indexBounds" : {

},
    "server" : "ANOC9:27017"
}>

```

## ***Support for sort Operations***

In MongoDB *sort* operations that sort documents based on an indexed field provide the greatest performance. Indexes in MongoDB, as in other databases, have an order: as a result, using an index to access documents returns results in the same order as the index.

A compound index needs to be created when sorting on multiple fields.

With compound indexes, the results can be in the sorted order of either the full index or an index prefix.

An index prefix is a subset of a compound index; the subset consists of one or more fields at the start of the index, in order.



*For example, given an index { a: 1, b: 1, c: 1, d: 1 }, the following subsets are index prefixes:*

{ a: 1 }, { a: 1, b: 1 }, { a: 1, b: 1, c: 1 }

A compound index can only help with sorting if it is a prefix of the sort.

For example, a compound index on “Age”, “Name” and “Class”

```
> db.testindx.ensureIndex({"Age": 1, "Name": 1,  
"Class": 1})
```

Will be useful for the following queries

```
> db.testindx.find().sort({"Age":1})  
> db.testindx.find().sort({"Age":1,"Name":1})  
> db.testindx.find().sort({"Age":1,"Name":1,  
"Class":1})
```

The above index won't be of much help in the following query

```
> db.testindx.find().sort({"Gender":1, "Age":1,  
"Name": 1})
```

You can diagnose how MongoDB defaults to processing by using the explain command.

## ***Unique index***

Creating index on a field doesn't ensure uniqueness thus if index is created on the *Name field* then two or more documents can have same names, however if uniqueness is one of the constraints that needs to be enabled then the *unique* property need to be set to *true* while creating the index. The below command will create a unique index on the *Name* field of the *testindx* Collection.

```
> db.testindx.ensureIndex({"Name":1},  
{"unique":true})
```

Now if try to insert duplicate names in the collection as shown below, MongoDB returns an error and does not allow insertion of duplicate records.

```
> db.testindx.insert({"Name":"uniquename"})  
> db.testindx.insert({"Name":"uniquename"})  
"E11000      duplicate      key      error      index:  
mydbpoc.testindx.$Name_1 dup key: { : "uniquename" }"
```

If you check the collection, you'll see that only the first "uniquename" was stored.

```
> db.testindx.find({"Name":"uniquename"})  
{ "_id" : ObjectId("52f4b3c3958073ea07f092ca"), "Name"  
: "uniquename" }  
>
```

Uniqueness can be enabled for compound indexes also which means that though individual fields can have duplicate values but the combination will always be unique.

For example if we have a unique index on {"name":1, "age":1},

```
> db.testindx.ensureIndex({"Name":1,      "Age":1},  
{"unique":true})  
>
```

Then the following inserts will be permissible

```
> db.testindx.insert({"Name":"usercit"})
```

```
> db.testindx.insert({"Name":"usercit", "Age":30})
```

However if we execute the below command

```
> db.testindx.insert({"Name":"usercit", "Age":30})
```

It'll throw an error

```
E11000      duplicate      key      error      index:  
mydbpoc.testindx.$Name_1_Age_1 dup key: { : "usercit",  
: 30.0 }
```

It might be the case at times that we create the collection and insert the documents first and then create an index on the collection, if we create a unique index on the collection which might have duplicate values in the fields on which the index is being created then the index creation will fail.

In order to cater to this scenario MongoDB provides “*dropDups*” option.

The “*dropDups*” option will save the first document found and remove any subsequent documents with duplicate values:

The following command will create a unique index on the “name” field and will delete any duplicate document if any.

```
>db.testindx.ensureIndex({"Name":1}, {"unique":true,  
"dropDups":true})
```

```
>
```

## ***system.indexes***

Whenever we create a database, by default a *system.indexes* collection is created. All of the information about a database's indexes is stored in the *system.indexes* collection. This is a reserved collection, so you cannot modify its documents or remove documents from it. You can manipulate it only through ***ensureIndex*** and the ***dropIndexes*** database command.

Whenever an index is created its Meta information can be seen in *system.indexes*. The following command can be used to fetch all the

index information about the mentioned collection

*db.collectionName.getIndexes()*

For example the below command will return all indexes created on the testindx Collection

```
> db.testindx.getIndexes()
[  
  {  
    "v" : 1,  
    "key" : {  
      "_id" : 1  
    },  
    "ns" : "mydbpoc.testindx",  
    "name" : "_id_"  
  },  
  {  
    "v" : 1,  
    "key" : {  
      "Name" : 1,  
      "Age" : 1  
    },  
    "unique" : true,  
    "ns" : "mydbpoc.testindx",  
    "name" : "Name_1_Age_1"  
  },  
  {  
    "v" : 1,  
    "key" : {  
      "Name" : 1  
    },  
    "unique" : true,  
    "ns" : "mydbpoc.testindx",  
    "name" : "Name_1",  
    "dropDups" : true
```

```
}
```

```
]
```

```
>
```

## ***dropIndex***

*dropIndex* command is used for removing the index.

The following command will remove the *Name* field index from the *testindx* collection.

```
> db.testindx.dropIndex({"Name":1})
```

```
{ "nIndexesWas" : 3, "ok" : 1 }
```

```
>
```

## ***reIndex***

When we have performed a number of insertion and deletion on the collection, at times it is required to rebuild the indexes so that the index can be used optimally. *reIndex* is used for rebuilding the indexes. The following command rebuilds all indexes on the collection in a single operation.

```
db.collectionname.reIndex()
```

This operation drops all indexes including the *\_id* index and then rebuilds all the indexes.

The following command rebuilds indexes of the *testindx* collection.

```
> db.testindx.reIndex()
```

```
{
```

```
  "nIndexesWas" : 2,
```

```
  "msg" : "indexes dropped for collection",
```

```
  "nIndexes" : 2,
```

```
  .....
```

```
  "ok" : 1
```

```
}
```

```
>
```

We will be discussing in details about the different types of indexes available in MongoDB in the next chapter.

## **How Indexing works**

MongoDB stores indexes as *BTree* structure hence range queries are automatically supported.

When there are multiple selection criteria in a query, MongoDB attempts to use one single best index to select a candidate set and then sequentially iterate through them to evaluate other criteria.

When handling a query the first time, MongoDB will create multiple execution plans (one for each available index) and let them take turns (within certain number of ticks) to execute until the fastest plan finishes. The result of the fastest executor will be returned and the system remembers the corresponding index used by the fastest executor.

Subsequent query will use the remembered index until certain number of updates has happened in the collection, then the system repeats the process to figure out what is the best index at that time.

*As collections change over time, the query optimizer deletes a query plan and reevaluates the query plan after any of the following events:*

1. *The collection receives 1,000 write operations.*
2. *The `reIndex` rebuilds the index.*
3. *You add or drop an index.*
4. *The `mongod` process restarts.*

*Note: If you want to override MongoDB's default index selection then the same can be done using `hint()` method.*

Since only one index will be used, it is important to look at the search or sorting criteria of the query and build additional composite index to match the query better.

Maintaining an index is not without cost as index needs to be updated when docs are created, deleted and updated, which incurs overhead for update operations.

*To maintain an optimal balance, we need to periodically measure the effectiveness of having an index (e.g. the read/write ratio) and delete less efficient indexes.*

# Part II Stepping Beyond the Basics

In the previous part we looked at how to create a database and perform basic operations on it such as search, update and delete. We also used selectors. Selectors give the users the ability to have much more fine grained control to find out the data that we really want.

In this part we will be covering advanced querying using conditional operators and regular expressions in the selector part. Each of these successively provides us with more fine-grained control over the queries we can write and, consequently, the information that we can extract from our MongoDB databases.

## Using Conditional Operators

Conditional Operators as the name implies are operators that refine the conditions that the query must match when extracting data from the database. We will be focusing on the following conditional operators

`$lt, $gt, $lte, $gte, $in, $nin and $not`

Let's look and understand each one in turn.



*The following example assumes a collection named Students that contains documents of the following prototype:*

```
{  
  _id: ObjectId(),  
  Name: "Full Name",  
  Age: 30,  
  Gender: "M",  
  Class: "C1",
```

**Score:** 95

}

We will first create the collection and insert few sample documents.

```
>db.students.insert({Name:"S1",Age:25,Gender:"M",
Class:"C1",Score:95})
>db.students.insert({Name:"S2",Age:18,Gender:"M",
Class:"C1",Score:85})
>db.students.insert({Name:"S3",Age:18,Gender:"F",
Class:"C1",Score:85})
>db.students.insert({Name:"S4",Age:18,Gender:"F",
Class:"C1",Score:75})
>db.students.insert({Name:"S5",Age:18,Gender:"F",
Class:"C2",Score:75})
>db.students.insert({Name:"S6",Age:21,Gender:"M",
Class:"C2",Score:100})
>db.students.insert({Name:"S7",Age:21,Gender:"M",
Class:"C2",Score:100})
>db.students.insert({Name:"S8",Age:25,Gender:"F",
Class:"C2",Score:100})
>db.students.insert({Name:"S9",Age:25,Gender:"F",
Class:"C2",Score:90})
>db.students.insert({Name:"S10",Age:28,Gender:"F",
Class:"C3",Score:90})
> db.students.find()
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name"
: "S1", "Age" : 25, "Gender" : "M", "Class" : "C1", "Score"
: 95 }

.....
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name"
: "S10", "Age" : 28, "Gender" : "F", "Class" : "C3",
"Score" : 90 }
>
```

## \$lt and \$lte

Let's start with `$lt` and `$lte` operators. They stand for “*Less Than*” and “*Less Than or Equal to*” operators respectively.

Let's say we want to *find all students with Age < 25*. Hence for this we will execute the following *find* with selector

```
> db.students.find({"Age": {"$lt": 25}})  
{ "_id" : ObjectId("52f8750ca13cd6a65998734e"), "Name" : "S2", "Age" : 18, "Gender" : "M", "Class" : "C1", "Score" : 85 }  
.....  
{ "_id" : ObjectId("52f87556a13cd6a659987353"), "Name" : "S7", "Age" : 21, "Gender" : "M", "Class" : "C2", "Score" : 100 }  
>
```

Next if we want to *find out all students with Age <= 25* then the following will be executed

```
> db.students.find({"Age": {"$lte": 25}})  
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25, "Gender" : "M", "Class" : "C1", "Score" : 95 }  
.....  
{ "_id" : ObjectId("52f87578a13cd6a659987355"), "Name" : "S9", "Age" : 25, "Gender" : "F", "Class" : "C2", "Score" : 90 }  
>
```

## \$gt and \$gte

These operators stand for “*Greater than*” and “*Greater than or equal to*” respectively.

Let’s *find out all the students with Age > 25*. This can be achieved by executing the following command

```
> db.students.find({"Age": {"$gt": 25}})  
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F", "Class" : "C3", "Score" : 90 }  
>
```

If we change the above example to return *students with Age>=25* then the command is as below

```
> db.students.find({"Age": {"$gte": 25}})  
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25, "Gender" : "M", "Class" : "C1", "Score" : 95 }  
.....  
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F", "Class" : "C3", "Score" : 90 }  
>
```

## \$in and \$nin

Let's find out all students who belong to either class C1 or C2. The command for the same is as below

```
> db.students.find({"Class": {"$in": ["C1", "C2"]}})  
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25, "Gender" : "M", "Class" : "C1", "Score" : 95 }  
.....  
{ "_id" : ObjectId("52f87578a13cd6a659987355"), "Name" : "S9", "Age" : 25, "Gender" : "F", "Class" : "C2", "Score" : 90 }  
>
```

The inverse of this can be returned by using *\$nin*.

Let's next find out students *who don't belong to class C1 or C2*. The command is as below

```
> db.students.find({"Class": {"$nin": ["C1", "C2"]}})  
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" : 28, "Gender" : "F", "Class" : "C3", "Score" : 90 }  
>
```

Let's next see how we can combine all the above operators and write a query.

Say we want to find out all students whose gender is either “M” or they belong to class “C1” or ‘C2” and whose age is greater than or equal to 25. This can be achieved by executing the following command

```
>db.students.find({$or:[{"Gender":"M","Class":{$in:["C1","C2"]}}, {"Age":{$gte:25}})}  
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25, "Gender" : "M", "Class" : "C1", "Score" : 95 }  
>
```

# Regular Expressions

In this section we will look at how we can use Regular expressions. Regular expressions are useful in scenarios where we want to *find say students with name starting with “A” etc.*

In order to understand this lets add 3-4 students more with different names.

```
> db.students.insert({Name:"Student1", Age:30,  
Gender:"M", Class: "Biology", Score:90})  
> db.students.insert({Name:"Student2", Age:30,  
Gender:"M", Class: "Chemistry", Score:90})  
> db.students.insert({Name:"Test1", Age:30,  
Gender:"M", Class: "Chemistry", Score:90})  
> db.students.insert({Name:"Test2", Age:30,  
Gender:"M", Class: "Chemistry", Score:90})  
> db.students.insert({Name:"Test3", Age:30,  
Gender:"M", Class: "Chemistry", Score:90})  
>
```

Say we want to find all students with names starting with “St” or “Te” and whose class begins with “Che”. The same can be filtered using regular expressions as shown below

```
> db.students.find({"Name":/(St|Te)/i,  
"Class":/(Che)/i})  
{ "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name"  
: "Student2", "Age" : 30, "Gender" : "M", "Class" :  
"Chemistry", "Score" : 90 }  
.....  
{ "_id" : ObjectId("52f89f06e451bb7a56e59089"), "Name"  
: "Test3", "Age" : 30, "Gender" : "M", "Class" :  
"Chemistry", "Score" : 90 }  
>
```

*In order to understand how the regular expression works let's take the query: We have mentioned “Name”:/(St|Te)/i.*

*//i* indicates that whatever we mention between this is a case insensitive regex.

**(St|Te)\*** indicates that the start of the Name string must be either “St” or “Te”.

The \* at the end means it will match anything after that.

When we put everything together, we are doing a case insensitive match of names that have either “St” or “Te” at the beginning of them. In the regex for the Class also the same Regex is issued.

Next let's complicate the query a bit by combining it with one of the conditional operators that we covered above.

Find out all Students with names as student1, student2 and who are male students with age >=25.

The command for this is as shown below.

```
>db.students.find({"Name":/(student*)/i,"Age":  
{"$gte":25}, "Gender":"M"})  
{ "_id" : ObjectId("52f89eb1e451bb7a56e59085"), "Name"  
: "Student1", "Age" : 30,  
"Gender" : "M", "Class" : "Biology", "Score" : 90 }  
{ "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name"  
: "Student2", "Age" : 30,  
"Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
```

## MapReduce

MapReduce is a process where the aggregation of data can be split up and executed across a cluster of computers to reduce the time that it takes to determine an aggregate result on a set of data. It's made up of two parts: Map and Reduce.

A more specific description:

 MapReduce is a framework for processing highly distributable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes use the

*same hardware) or a grid (if the nodes use different hardware). Computational processing can occur on data stored either in a file system (unstructured) or in a database (structured).*

*“Map” step: The master node takes the input, partitions it up into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.*

*“Reduce” step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.*

In order to understand how it works let's consider a small example, where we will *find out the number of male and female students* in our collection.

This involves the following steps: first we will create the *map* and *reduce* functions and then we will call the *mapReduce* function and pass the necessary arguments.

Let's start with defining the *map* function

```
> var map = function(){emit(this.Gender,1);};  
>
```

This step takes as input the document and based on the “Gender” field it emits documents of the type {"F", 1} or {"M", 1}.

Next we will create the *reduce* function

```
> var reduce = function(key, value){return  
  Array.sum(value);};  
>
```

This will group the documents emitted by the map function on the key field which in our example is “Gender” and return the sum of values which in the above example is emitted as “1”. Hence the output of the reduce function defined above is gender wise count.

Finally we will put them together using the *mapReduce* function as follows

```

> db.students.mapReduce(map,      reduce,      {out:
"mapreducecount1"})
{
  "result" : "mapreducecount1",
  "timeMillis" : 280,
  "counts" : {
    "input" : 15,
    "emit" : 15,
    "reduce" : 2,
    "output" : 2
  },
  "ok" : 1,
}
>

```

This actually is applying the *map*, *reduce* function which we defined on the *students* collection. The final result is stored in a new collection called *mapreducecount1*.

In order to vet we will run *find()* command on the *mapreducecount1* collection as shown below.

```

> db.mapreducecount1.find()
{ "_id" : "F", "value" : 6 }
{ "_id" : "M", "value" : 9 }
>

```

We will use one more example to explain the working of MapReduce. We will next use MapReduce to *find out Class wise Average Score*. So as we saw in the above example, we need to create first the *map* function and then the *reduce* function and finally we will combine them to store the output in a collection in our database. The code snippet is as shown below

```

> var map_1 = function()
{emit(this.Class,this.Score);};

```

```

> var reduce_1 = function(key, value){return
Array.avg(value);};
>db.students.mapReduce(map_1,reduce_1,
out:"MR_ClassAvg_1")
{
  "result" : "MR_ClassAvg_1",
  "timeMillis" : 57,
  "counts" : {
    "input" : 15, "emit" : 15,
    "reduce" : 3 , "output" : 5
  },
  "ok" : 1,
}
> db.MR_ClassAvg_1.find()
{ "_id" : "Biology", "value" : 90 }
{ "_id" : "C1", "value" : 85 }
{ "_id" : "C2", "value" : 93 }
{ "_id" : "C3", "value" : 90 }
{ "_id" : "Chemistry", "value" : 90 }
>

```

The first step is defining the *map* function which loops through the collection documents and returns output as {“Class”: Score} for e.g. {“C1”:95}.

The second step does grouping on the class and computes Average of the scores for that class.

The Third steps combine the results, it defines the collection to which the *map*, *reduce* function need to be applied and finally it is defining where to store the output which in this case is a new collection *MR\_ClassAvg\_1*.

In the last step we use *find* in order to check the resulting output.

## aggregate()

In the previous section we covered an introduction of MapReduce function. In this section we will take a glimpse of the Aggregation framework of MongoDB which provides a means to compute aggregated values without having to use the MapReduce function.

We will depict the above two discussed outputs using the *aggregate* function. First output was to *find count of male and female students*. So the same can be achieved by executing the following command

```
> db.students.aggregate({$group:{_id:"$Gender",
totalStudent: {$sum: 1}}})
{
  "result": [
    {
      "_id" : "F",
      "totalStudent" : 6
    },
    {
      "_id" : "M",
      "totalStudent" : 9
    }
  ],
  "ok" : 1
}
>
```

Similarly in order to *find out Class wise Average score* the following command can be executed

```
>      db.students.aggregate({$group:{_id:"$Class",
AvgScore: {$avg: "$Score"}}})
{
  "result" : [ {
    "_id" : "Biology",
    "AvgScore" : 90
  },
  .....
  {
    "_id" : "C1",
    "AvgScore" : 85
  }
], "ok" : 1}
>
```

# Part III

In the previous sections we covered how to get started with the MongoDB database. In this section we will look at how to design data model for an application. The MongoDB database provides two options for designing a data model i.e. the user can either embed related objects within one another or can reference each other using ID. In this section we will explore these options.

In order to understand this we will design a blogging application and demonstrate the usage of the two options.



*A typical blog application consists of the following scenarios:*

*We have People posting Blogs on different subjects, in addition to the subject categorization different tags can also be used. As an example if the category is politics and the post talks about a politician in general then that politician's name can be added as a tag to the post, which helps the users not only in finding out posts related to their interest quickly but also enables them to link related posts together.*

*The people viewing the blog can comment on the blog posts.*

## Relational Data Modeling and Normalization

Before jumping into MongoDB's approach we'll take a little detour into how we will model this in relational databases such as SQL.

In relational databases the data modeling typically progresses by defining the tables and gradually removing data redundancy to achieve a Normal form.

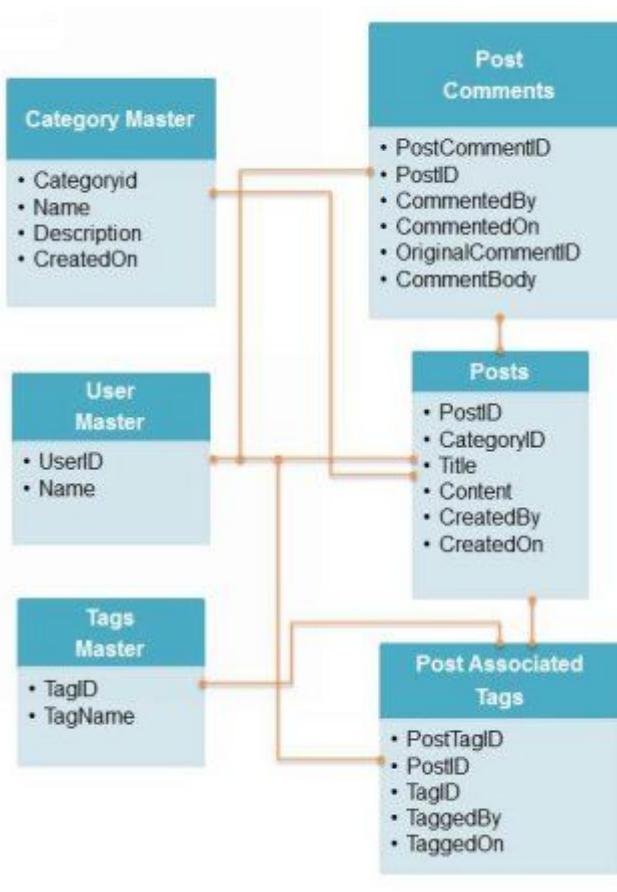
### **What Is a Normal Form?**

In relational databases Normal form typically begins by creating tables as per the application requirement and then gradually

removing redundancy to achieve the highest normal form which is also termed as the Third Normal Form or 3NF. In order to understand this better let's try and put in the Blogging Application data in tabular form. The initial data might be of the following form

Author	Posts	Category	Tag	Comments	Commenter

This data is actually in First Normal form. Will have lots of redundancy as we can have multiple comments against the Posts and multiple Tags can be associated with the post. Hence data will be redundant. The problem with redundancy, of course, is that it introduces the possibility of inconsistency, where various copies of the same data may have different values. To remove this redundancy, we need to further normalize the data by splitting it into multiple tables. As part of this step, we must identify a *key* column which uniquely identifies each row in the table so that we can create links between the tables. Hence the above scenarios when modeled using the 3NF normal forms will look like as depicted below:



**Fig 6-2: RDBMS Diagram**

In this case, we have a data model that is free of redundancy, allowing us to update without having to worry about updating multiple rows. In particular, we no longer need to worry about *inconsistency* in the data model.

## **Problem with the normal forms**

As already mentioned, the nice thing about normalization is that it allows for easy updating without any redundancy i.e. it helps in keeping the data consistent. Updating a user name will just need to update the name in the Users table.

However the problem arises when we try to get the data back *out*. For instance if we have to find out all tags and comments associated with the posts by a specific user, the relational database programmer uses a JOIN. Though using JOINS the database returns all data as

per the application screen design but the real problem is what operation the database performs to get that result set.

Generally an RDBMS reads from a disk and does seek which takes well over 99% of the time spent reading a row. When it comes to disk access, random seeks are the enemy. The reason why this is so important in this context is because JOINs typically require random seeks. JOIN operation is one of the most expensive operations within a relational database. Additionally, if you end up needing to scale your database to multiple servers, you introduce the problem of generating a *distributed join*, a complex and generally slow operation.

# MongoDB Document Data Model Approach

We have already seen that in MongoDB, data is stored in *documents*. Fortunately for us as application designers, that opens up some new possibilities in schema design. Unfortunately for us, it also complicates our schema design process. There is no longer a “garden path” of normalized database design to go down, and the go-to answer when faced with general schema design problems in MongoDB is “it depends.”

If we have to model the above using MongoDB document model then we might store the blog data in a document as follows

```
{  
  "_id" : ObjectId("508d27069cc1ae293b36928d"),  
  "title" : "This is the title",  
  "body" : "This is the body text.",  
  "tags" : [  
    "chocolate",  
    "spleen",  
    "piano",  
    "spatula"  
  ],  
  "created_date" : ISODate("2012-10-28T12:41:39.110Z"),  
  "author" : "Author 1",  
  "category_id" : ObjectId("508d29709cc1ae293b369295"),  
  "comments" : [  
    {  
      "subject" : "This is comment 1",  
      "body" : "This is the body of comment 1."  
    }  
  ]}
```

```

    "author" : "author 2",
    "created_date":ISODate("2012-10-28T13:34:23.929Z")
}
]}

```

As we can see we have embedded the comments and Tags within a single document only, alternatively we could “normalize” the model a bit by referencing the comments and tags by its \_id field:

```

// Authors document:
{
  "_id": ObjectId("508d280e9cc1ae293b36928e"),
  "name": "Jenny",}
// Tags document:
{
  "_id": ObjectId("508d35349cc1ae293b369299"),
  "TagName": "Tag1",.....}
// Comments document:
{
  "_id": ObjectId("508d359a9cc1ae293b3692a0"),
  "Author": ObjectId("508d27069cc1ae293b36928d"),
  .....
  "created_date" : ISODate("2012-10-28T13:34:59.336Z")
}
//Category Document
{
  "_id": ObjectId("508d29709cc1ae293b369295"),
  "Category": "Catgeory1".....}
//Posts Document
{
  "_id" : ObjectId("508d27069cc1ae293b36928d"),
  "title" : "This is the title","body" : "This is the body
text.",
```

```

"tags" : [ ObjectId("508d35349cc1ae293b369299"),
ObjectId("508d35349cc1ae293b36929c")
],
"created_date" : ISODate("2012-10-28T12:41:39.110Z"),
"author_id" : ObjectId("508d280e9cc1ae293b36928e"),
"category_id" : ObjectId("508d29709cc1ae293b369295"),
"comments" : [
ObjectId("508d359a9cc1ae293b3692a0"),
]}

```

The remainder of this chapter is devoted in identifying which solution will work in our context i.e. whether to use referencing or whether to embed.

## ***Embedding***

In this section we will look at the possible causes when embedding will have positive impact on the performance.

Embedding can be useful when we want to fetch some set of data together and display on the screen together. For example in the above case say we have a page which is displaying comments associated with the blog together in that case the comments can be embedded in the *Blogs* document.

The benefit of this approach is that since MongoDB stores the documents contiguously on disk all the related data can be fetched in a single seek. Apart from this since JOINS are not supported and we had used referencing in this case then the application might do something like the following to fetch the comments data associated with the blog.

1. Fetch the associated *comments \_id* from the *blogs* document
2. Fetch the *comments* document based on the *comments\_id* found in the first step.

If we take this approach, not only does the database have to do multiple seeks to find our data, but also additional latency is introduced into the lookup since it now takes *two* round-trips to the database to retrieve our data.

Hence if the application frequently accesses the comments data along with the blogs then almost certainly embedding the comments within the *blog* documents will have a positive impact on the performance.

Another concern that weighs in favor of embedding is the desire for *atomicity* and *isolation* in writing data. MongoDB is designed without multi documents transaction i.e. MongoDB only provides atomic operations on the level of a single document hence data that need to be updated together atomically needs to be placed together in a single document.

When we update data in our database, we want to ensure that our update either succeeds or fails entirely, never having a “partial success,” and that any other database reader never sees an incomplete write operation.

## **Referencing**

We have seen that embedding is the approach that will provide the best performance in many cases and also provides data consistency guarantees. However, in some cases, a more normalized model works better in MongoDB.

One reason for having multiple collections and adding references is the increased flexibility it gives when querying the data. Let's understand this with the blogging example we mentioned above.

We have already seen how the schema will be when we use embedded schema which will work very well when displaying all the data together on a single page i.e. the page which will display the Blog Post followed by all the associated comments.

Now suppose we have a requirement to search for the comments posted by a particular user, the query (using this embedded schema) would be as follows:

```
db.posts.find({'comments.author': 'author2'}, {'comments': 1})
```

The result of this query, then, would be documents of the following form:

```
{
  "_id" : ObjectId("508d27069cc1ae293b36928d"),
  "comments" : [ {
    "subject" : "This is coment 1",
    "body" : "This is the body of comment 1.",
    "author_id" : "author2",
    "created_date" : ISODate("2012-10-28T13:34:23.929Z")}...]
}
{
  "_id" : ObjectId("508d27069cc1ae293b36928d"),
  "comments" : [
    {
```

```

"subject" : "This is comment 1",
"body" : "This is the body of comment 1.",
"author_id" : "author2",
"created_date" : ISODate("2012-10-28T13:34:23.929Z")
}...]}

```

The major drawback to this approach is that we get back *much* more data than we actually need. In particular, we can't ask for just author2's comments; we have to ask for posts that author2 has commented on, which includes all the other comments on those posts as well. Hence this data will require further filtering within the application code.

On the other hand, suppose we decide to use a normalized schema. In this case we will have three documents i.e. "Authors", "Posts" and "Comments".

The "Authors" document will have Author specific contents such as Name, Age, Gender etc., "Posts" document will have Posts specific details such as Post creation time, Author of the post, actual content and subject of the post.

The "Comments" document will have the Post's comments such as Commented On date time, created by author and the text of the comment. The same is depicted as below:

```

// Authors document:
{
  "_id": ObjectId("508d280e9cc1ae293b36928e"),
  "name": "Jenny",
  .....
}
//Posts Document
{
  "_id"      :      ObjectId("508d27069cc1ae293b36928d"),
  .....
}

```

```
// Comments document:  
{  
  "_id": ObjectId("508d359a9cc1ae293b3692a0"),  
  "Author": ObjectId("508d27069cc1ae293b36928d"),  
  "created_date" : ISODate("2012-10-28T13:34:59.336Z"),  
  "Post_id": ObjectId("508d27069cc1ae293b36928d"),  
  .....  
}
```

In this scenario the query of finding the comments by say author “author2” can be satisfied by a simple `find()` on the `comments` collection:

```
db.comments.find({"author": "author2"})
```

In general, if your application’s query pattern is well-known and data tends to be accessed in only one way, an embedded approach works well. Alternatively, if your application may query data in many different ways, or you are not able to anticipate the patterns in which data may be queried, a more “normalized” approach may be better.

For instance, in the above schema, we will be able to sort the comments or return a more restricted set of comments using `limit`, `skip` operators. Whereas in the embedded case we’re stuck retrieving all the comments in the same order in which they are stored in the post.

Another factor that may weigh in favor of using document references is when you have one-to-many relationships.

For instance, a popular blog with a large amount of reader engagement may have hundreds or even thousands of comments for a given post. In this case, embedding carries significant penalties with it:

**Effect on Read Performance** - As the document size increase it will occupy more RAM. The problem with RAM is that a MongoDB database caches frequently accessed documents in RAM and larger the documents become lesser is the probability of it fitting in RAM and hence it will lead to more page faults while retrieving the documents and hence will lead to random disk i/o which will further slowdown the performance.

**Effect on Update performance** - As the size increases and an update operation is performed on such documents to append data eventually MongoDB is going to need to move the document to an area with more space available. This movement, when it happens, significantly slows update performance.

Apart from this the MongoDB documents have a hard size limit of 16 MB. Although this is something to be aware of, you will usually run into problems due to memory pressure and document copying well before you reach the 16 MB size limit.

One final factor that weighs in favor of using document references is the case of many-to-many or M:N relationships.

For instance in our above example we have Tags, Each Blog can have multiple tags and each Tag can be associated to multiple Blog entries.

One approach to implement the Blogs-Tags M:N relationship is to have the following three collections

1. Tags Collection which will store the Tags Details
2. Blogs collection which will have Blogs Details
3. Third Collection Tag-To-Blog Mapping will have mapping between the tags and the blogs.

This approach is similar to the one we have in the relational databases but this will negatively impact the application performance as the queries will end up doing a lot of application-level “joins”.

Alternatively, we can use embedding model where we will embed the Tags within the Blogs document, but this will lead to data duplication. Though this will simplify the read operation a bit but will increase the complexity of the update operation, as while updating a tag detail the user needs to ensure that the updated tag is updated at each and every place where it has been embedded in other blogs documents.

Hence for many-to-many joins, a compromise approach is often best, embedding a list of `_id` values rather than the full document:

```
// Tags document:  
{  
  "_id": ObjectId("508d35349cc1ae293b369299"),  
  "TagName": "Tag1",  
  .....  
}
```

```

// Posts document with Tag IDs added as References
//Posts Document
{
  "_id" : ObjectId("508d27069cc1ae293b36928d"),
  "tags" : [
    ObjectId("508d35349cc1ae293b369299"),
    ObjectId("508d35349cc1ae293b36929a"),
    ObjectId("508d35349cc1ae293b36929b"),
    ObjectId("508d35349cc1ae293b36929c")
  ],
  .....
}

```

Though querying will be a bit complicated but we no longer need to worry about updating a tag everywhere.

In summary Schema design in MongoDB is one of the very early decisions that we need to take and is dependent on the application requirements and queries.

As we have seen above when we need to access the data together or we need to make atomic updates then using embedding will have a positive impact, however if we need more flexibility while querying or if we have a Many to Many relationships then using references will be a worthy decision.

Ultimately, the decision depends on the access patterns of your application, and there are no hard-and-fast rules in MongoDB, hence on basis of the access pattern the data model need to be thought of and decided. In the next section we will be covering various data modeling considerations.

## ***Data modeling decisions***

This involves determining how to structure the documents to model the data effectively.



The most important decision involved in whether to embed the data or add reference to the data i.e. use references.

*This point is best demonstrated with an example. Suppose we have a book review site which will have authors and books as well as*

*reviews with threaded comments.*

Now the question is how should be structure the collections.

*The decisions lies in the use cases i.e. it depends on the number of comments expected on per book and how frequently the read vs. write operations will be performed.*

## ***Operational considerations***

In addition to the way the elements interact with each other i.e. whether to store the documents in an embedded manner or use references, a number of other operational factors are important while designing a data model for the application. These factors include the following:

### **Data Lifecycle management**

This feature needs to be used if our application has datasets which need to be persisted in the database only for a limited time period.

*Say for e.g. in our above example if we need to retain the data related to the review and comments for say a month then this feature can be taken into consideration.*

This is implemented by using the Time to live feature of the collection.

The Time to live or the TTL feature of the collection expires documents after a period of time.

Additionally if the application requirement is to work with only the recently inserted documents then using Capped collections will help optimize the performance.

### **Indexes**

Indexes can be created to support commonly used queries to increase the performance.

*By default an index is created by MongoDB on the \_id field.*

Few points which we need to take into consideration while creating indexes are:

1. Each index requires at least 8KB of data space.
2. Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive as each insert must add keys to each index.
3. Collections with high proportion of read operations to write operations often benefit from additional indexes. Indexes do not affect un-indexed read operations.

## Sharding

Among the various factors one of the important factors while designing the application model is whether to partition the data or not. This is implemented using Sharding in MongoDB.

Sharding is also referred as Partitioning of data.

In MongoDB sharding involves partitioning a collection within a database to distribute the collections documents across a cluster of machines which are also termed as Shards. This can have significant impact on the performance. We will discuss more about sharding in the MongoDB Explained chapter.

## Large number of collections

The design considerations for having multiple collections versus storing data in a single collection are as below:

- There is no performance penalty in choosing multiple collections for storing data.
- Having distinct collections for different types of data can have performance improvements in high throughput batch processing applications.

When using models that have a large number of collections, we need to consider the following behaviors:

- Each collection has a certain minimum overhead of a few kilobytes.
- Each index, including the index on `_id`, requires at least 8KB of data space.

As mentioned above in the way data is stored in MongoDB we know that there's a single `<database>.ns` file which stores all meta-data for each [database](#).

*Each index and collection has its own entry in the namespace file, so we need to consider the limits on the size of namespace files in MongoDB while thinking of implementing large number of collections.*

## Document growth

Certain updates to documents can increase the document size, such as pushing elements to an array and adding new fields. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. This internal relocation can be both time and resource consuming. Although MongoDB automatically provides padding to minimize the occurrence of relocations, you may still need to manually handle document growth.

# Summary

In this chapter we covered the basic CRUD operator's followed with advanced querying capabilities. Finally the chapter is concluded with the understanding of the two ways of storing and retrieving data i.e. Embedding and Referencing.

In the following chapter we will cover MongoDB Architecture, its core components and features.

# MongoDB Explained

*“MongoDB explained covers deep dive architectural concepts of MongoDB.”*

In this chapter we will learn about MongoDB architecture wherein we will cover Core Processes and Tools, Standalone deployment, Sharding concepts, Replication concepts and Production Deployment.

We will discuss about High Availability considerations in deployment of MongoDB. We will cover how data is stored under the hood and how writes happens using Journaling. Finally we will touch upon GridFS, different types of indexes available in MongoDB.

# Architecture

In this section we will look at the deployment architecture components. We will discuss about the core processes that are packaged as MongoDB. Then we will see how the data is actually stored.

## Core processes

The core components in the MongoDB package are:

- **mongod** which is the core database process
- **mongos** which is the controller and query router for sharded clusters
- **mongo** which is the interactive MongoDB shell

In the installation chapter we saw that all these are available as applications under the bin folder.

Let's discuss these components in detail.

### ***mongod***

mongod is the primary daemon process for the MongoDB system. This is the process which handles data requests, manages data format and performs background management operations.

We have already seen in the installation chapter that when we run mongod with no arguments, it uses the default data directory (/data/db or C:\data\db on Windows) and port (27017) where the process listens for sockets connection.

It is important to create the data directory and to ensure that the user has write permissions on the directory before we start the Mongod process.

If the directory doesn't exist or the user doesn't have write permissions on the directory in both the cases the start of this

process will fail. The Server will also fail to start if the port is not available.

mongod also sets up a very basic HTTP server that listens on port 1000 higher than the main port which in this case is 28017. This basic HTTP server provides us administrative information about our database on the URL <http://localhost:28017>

## ***mongo***

Mongo is an interactive JavaScript shell interface to MongoDB, which provides a powerful interface for systems administrators as well as a way for developers to test queries and operations directly with the database from the command line. When you start up the mongo shell it connects to the test database on MongoDB server and assigns the database connection to the global variable called db.

As a developer or administrator you need to change the database from test to your database post the first connection is made. You can do this by *use <dbname>*

## ***mongos***

Mongos stands for “MongoDB Shard” is a routing service for MongoDB shard configurations that processes queries from the application layer and determines the location of data in the sharded cluster. We will discuss Mongos in more detail in the sharding section, right now the readers can think of mongos as the process which routes the queries to the correct server holding the data.

# **MongoDB Tools**

Apart from the core services there are various tools that are available as part of the MongoDB installation.

- *mongodump*: This utility is used as part of an effective backup strategy. This is used for creating a binary export of the contents of the database.

- ***mongorestore***: The binary database dump which is created by the mongodump utility is imported to a new or an existing database using mongorestore utility.
- ***bsondump***: This utility converts the BSON files into human-readable format such as JSON, CSV. For example this utility can be used for reading the output file generated by mongodump.
- ***mongoimport, mongoexport***: mongoimport provides a method for taking data in [JSON](#), [CSV](#), or [TSV](#) and importing it into a mongod instance. Mongoexport provides a method to export data from a mongod instance into JSON, CSV, or TSV.
- ***mongostat, mongotop, mongosniff***: These utilities provide diagnostic information related to the current operation of a mongod instance.

## Standalone deployment

This is used for development purpose and this doesn't ensure any redundancy of data and neither ensures recovery in case of failures. Hence it's not recommended to be used in production environment. Standalone deployment has the following components: A single mongod and a Client connecting to the mongod.



*Fig 7-1: Standalone deployment*

📍 *MongoDB uses Sharding and Replication to provide a highly available system by distributing and duplicating the data. In the coming sections we will look at Sharding and Replication. Following*

*that we'll look at the recommended production deployment architecture.*

## Replication

In a standalone deployment if the mongod goes down then all the data will be lost which will not be acceptable in a production environment. Hence replication is used to offer safety against such kind of data loss.

Replication provides for data redundancy by replicating data on different nodes and thus provides protection of data in case of node failure. Replication provides high availability in a MongoDB deployment.

Replication also simplifies certain administrative tasks where the routine tasks such as backups can be offloaded to the replica copies, freeing the main copy to handle the important application requests.

In some scenarios it can also help in scaling the reads by enabling the client to read from the different copies of data.

In this section we will discuss how replication works in MongoDB, what are the various components of the same.

There are two types of replication supported in MongoDB

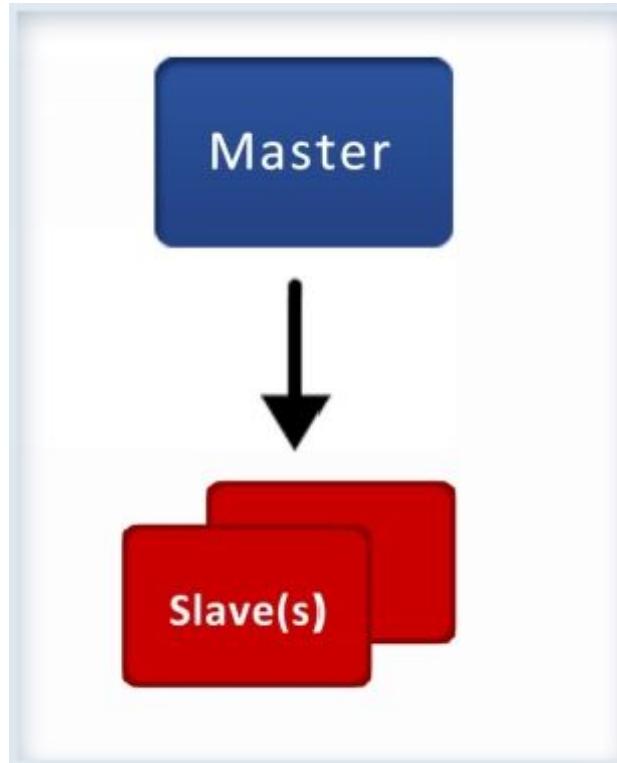
- Traditional Master/Slave Replication
- Replica set

### ***Master/Slave Replication***

In MongoDB the traditional master/slave replication is available but it is recommended only for more than 12 node replications. The preferred replication approach is replica sets which we will explain in a while. In this type of replication there is one master and number of slaves which replicate data from the master. The only advantage with this type of replication is that there is no explicit limit on the number of slaves in a cluster. However having thousands of slaves will overburden the master node hence in practical scenarios it's better to have less than dozen slaves. In addition to this the master-

slave replication provides less redundancy, and does not automate failover.

In a basic master/slave setup we have two types of mongod instances: one instance will be in master mode and the remaining in slave mode. Since the slaves are replicating from the master hence all slaves need to be aware of the master's address.



*Fig 7-2: Master Slave Replication*

The master node maintains a capped collection (oplog) which stores an ordered history of logical writes to the database.

The slaves replicate the data using this oplog collection. Since the oplog is a capped collection if a slave falls too far behind the master's state then the slave may become out of sync hence in that scenario the replication will stop, and manual intervention will be needed to re-establish the replication.

*There are two main reasons behind a slave becoming out of sync:*

- Slave shuts down or stops and restarts later. During this time oplog may have deleted the log of operations required to be applied on the slave.*

- Slave is slow in executing the updates that are available from the master.

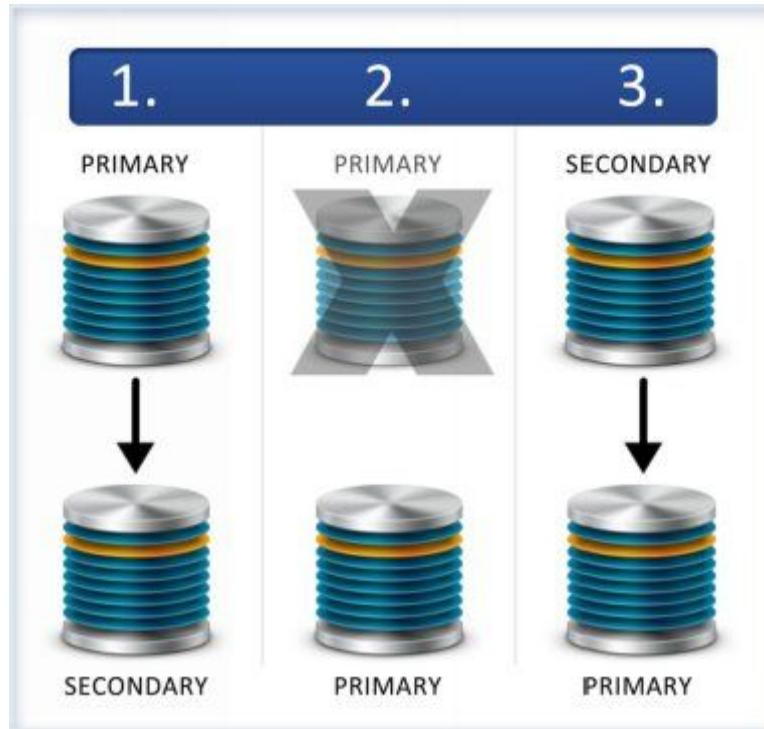
## **Replica Set**

Replica set is a more sophisticated form of traditional master-slave replication and is a recommended method in MongoDB deployments.

Replica sets are basically a type of master-slave replication but provides automatic failover. A Replica set has one master which is termed as Primary and multiple slaves which are termed as secondary in the Replica set context however unlike master-slave replication there's no one node which is fixed to be Primary in replica set.

In case a master goes down in replica set, one of the slaves is automatically promoted to become the master. Thus the clients will now connect to the new master and data and application will remain available. In replica set this failover happens in an automated fashion. We will explain the details of how this process happens in detail a little later.

The primary node is selected through an election mechanism by the cluster and it changes to another node in case the Primary goes down.



**Fig 7-3: Two members Replica Set Failover**

Let's discuss the various steps that happen for a two member Replica Set in failover.

1. Primary Goes down, Secondary is Promoted as Primary
2. Original Primary Comes up, it acts as Slave and becomes the Secondary node.

Hence the points to be noted are:

- Replica set is a cluster of mongod's which replicate among one another and ensure automatic failover.
- One mongod will be the primary member of the replica set and the others will be secondary members.
- The Primary member is elected by the members of the replica set. All writes are directed to the primary member whereas the secondary members replicate from the primary asynchronously using oplog.
- The secondaries data sets reflect the primary data sets enabling them to be promoted as primary in case of unavailability of the current primary.

*Replica Set replication has one limitation that a replica set can have maximum of up to 12 members only and at any given point of time in a 12-member replica set only 7 can participate in a vote. We will explain the voting concept in replica set in detail.*

## **Primary and Secondary Members**

Before we move ahead and look at how the replica set functions we will briefly look at the type of members that a Replica set can have. We have primarily two types of members

- Primary member
- Secondary member

### **Primary Member**

A Replica set can have only one primary which is elected by the voting nodes in the replica set. Any node with associated priority as 1 can be elected as a primary. The client redirects all the write operations to the primary member which is then later replicated to the secondary members.

## **Secondary Member**

A normal secondary member holds the copy of the data. The secondary member can vote and also can be a candidate for being promoted as primary in case of failover of current primary.

In addition to this a replica set can have other types of secondary members also.

## **Types of Secondary Members**

**Priority 0 Members** – These are the secondary members which maintain a copy of the primary's data but can never become a primary in case of a failover.

Apart from that they function as a normal secondary node and they can participate in voting and can accept read requests. The Priority 0 members are created by setting the priority to 0.

*Such types of members are specifically useful for the following reasons:*

1. *They can serve as a cold standby*
2. *In replica sets which has varied hardware or geographic distribution this configuration ensures that only the qualified members get elected as primary.*
3. *In replica set which spans across multiple data centers across network partitioning, this configuration can help ensure that the main data center has the eligible primary. This is used to ensure the failover is quick.*

**Hidden Members** – Hidden members are the 0-priority members which are hidden from the client applications.

Like the 0-priority members this members also maintains copy of the primary's data, cannot become primary and can participate in the voting but unlike 0-priotiy members they don't serve any read requests or receives any traffic beyond what replication requires.

A node can be set as hidden member by setting the hidden property to true.

*In the replica set these members can be useful for dedicating them to reporting needs or to do backups.*

**Delayed Members** – Delayed members are the secondary members which replicate data from the primary's oplog with a delay.

These members may help recover from various kinds of human errors such as accidentally dropped databases or errors that were caused due to unsuccessful application upgrades.

While deciding on the delay time; we need to consider our maintenance windows period and the size of the oplog. The delay time should be either equal to or greater than the maintenance window. And the oplog size is to be set in a manner to ensure that no operations are lost while replicating.

*We need to note that since the delayed members will not have up-to-date data as the primary node hence the priority should be set to 0 so that they cannot become primary and also the hidden property should be true in order to avoid any read requests from the users.*

**Arbiters** – This are the secondary members which do not hold copy of primary's data hence they can never become primary. They are solely used as member for participating in [voting](#).

This enables the replica set to have an uneven number of nodes without incurring any replication cost which arises with data replication.

**Non-Voting Members** – Non-voting members hold copy of the primary's data and can accept read operations from client and can become primary but they cannot vote in an election.

The voting ability of a member can be disabled by setting its votes to 0. By default every member has one vote.

*Say for e.g. we have a replica set with 7 members then using the following sequence of commands in mongo shell the votes for fourth, fifth and sixth member are set to 0.*

```
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[4].votes = 0
cfg.members[5].votes = 0
rs.reconfig(cfg)
```

*Though this setting allows the fourth, fifth and sixth member to be elected as primary but while voting there vote will not be counted. They become non-voting members which implies they can stand for election but cannot vote themselves.*

We will see how the members can be configured in the “implementing advanced clustering with replica sets” section.

## Elections

In this section of the chapter we will look at the process of election for selecting a Primary Member.

*A server has to get a majority of the total votes to be elected, not just a majority. Say for e.g. if we have X servers and each server has 1 vote then a server needs at least  $[(X/2) + 1]$  votes to become primary.*

If a server gets the required number of votes or more then it will become primary.

*The original primary is still part of the set. If you bring it back up, it will become a secondary server (until it gets the majority of votes again).*

One complication with this voting system is that you can't just have a master and a slave.

Reason being if you just set up a master and a slave, the system has a total of 2 votes, so a server needs both votes to be elected master because 1 is not a majority.

In case one of the two servers goes down, the other server only has 1 out of 2 votes, so it will become (or stay) a slave.

In case the network is partitioned, suddenly the master doesn't have a majority of the votes (it only has its own 1 vote), so it'll be demoted to a slave. The slave also doesn't have a majority of the votes, so it'll stay a slave (so you'd end up with two slaves until the servers can reach each other again).

To avoid such a situation Replica sets have a number of ways of avoiding this situation.

One of the simplest and most versatile ways is using an Arbiter which will help in resolving such disputes. It's just a voter and can even be on the same machine as another server as it's very lightweight.

So, let's say we set up a master, a slave, and an arbiter, each with 1 vote (total of 3 votes). Then, if we have the master and arbiter in one data center and the slave in another, if a network partition occurs, the master still has a majority of votes (master + arbiter). The slave only has 1 vote. If the master fails and the network is not partitioned, the arbiter can vote for the slave, promoting it to master.

With this three-server setup, we get a robust failover deployment.

### **Example - Working of Election Process in More Details**

In this section we will look in more details and understand how actually the election happens.

Let's say we have a replica set with 3 members: X, Y, and Z.

Every two seconds, each server sends out a heartbeat request to the other members of the set.

So, if we wait a few seconds, X sends out heartbeats to Y and Z. They respond with information about their current situation such as the state they're in (primary/secondary), if they are eligible to become primary, their current clock time, etc. X receives this info and updates its "map" of the set: if members have come up or gone down, changed state, and how long the round trip took.

At this point, if X's map changes, X will check a couple of things:

- If X is primary and a member went down, it will make sure it can still reach a majority of the set. If it cannot, it'll demote itself to a secondary.

*Demotions - There is one problem with X demoting itself: in MongoDB, writes default to fire-and-forget. Thus, if people are doing fire-and-forget writes on the primary and it steps down, they might not realize X is no longer primary and keep sending writes to it and since the writes don't get a response on the client, the client wouldn't know. Hence it's recommended to use safe writes. So, when a primary is demoted, it also closes all connections to clients so that they will get a socket error when they send the next message. All of the client libraries know to re-check who is primary if they get an error. Thus, they'll be able to find which the new primary is and not accidentally send an endless stream of writes to a secondary.*

- If X is a secondary, it'll occasionally check if it should elect itself, even if its map hasn't changed.
  - First, it'll do a sanity check such as does another member think its primary? Does X think it's already primary? Is X ineligible for election? If it fails any of the basic questions, it'll continue idling along as is.
  - If it seems as though a new primary is needed, X will proceed to the first step in election:
    - It sends a message to Y and Z which are the other members of the set, telling them "I am considering running for primary, can you advise me on this matter?"
    - When Y and Z get this message, they quickly check their world view. Do they already know of a primary? Do they have more recent data than X? Does anyone they know of have more recent data than X? They run through a huge list of sanity checks and, if everything seems satisfactory, they tentatively reply "go ahead." If they find a reason that X cannot be elected, they'll reply "stop the election!"
    - If X receives any "stop the election!" messages, it cancels the election and goes back to life as a secondary.
    - If everyone says "go ahead," X continues with the second (and final) phase of the election process.
  - For the second phase,
    - X sends out a second message that is basically, "I am formally announcing my candidacy."

- At this point, Y and Z make a final check: do all of the conditions that held true before still hold?
- If so, they allow X to take their election lock and send back a vote. The election lock prevents them from voting for another candidate for 30 seconds.
- If one of the checks doesn't pass the second time around (fairly unusual, at least in 2.0), they send back a veto.

- If anyone veto, the election fails. Suppose that Y votes for X and Z veto X. At that point, Y's election lock is taken, it cannot vote in another election for 30 seconds. That means that, if Z wants to run for primary, it better be able to get X's vote. That said, it should be able to if Z is a viable candidate.
- If no one veto and the candidate member receives votes from a majority of the set, the candidate becomes primary.

 *The priority setting affects elections. Members will prefer to vote for members with the highest priority value. Members with a priority value of 0 cannot become primary and do not seek election. A replica set does not hold an election as long as the current primary has the highest priority value and is within 10 seconds of the latest oplog entry in the set. If a higher-priority member catches up to within 10 seconds of the latest oplog entry of the current primary, the set holds an election in order to provide the higher-priority node a chance to become primary.*

## Data Replication Process

In this section we will look at how the data is replicated among the members of the replica set.

The members of replica set replicate data continuously. The members continuously record the operations they are performing on the data sets in its Oplog which is a capped collection. The primary node maintains its oplog which the secondary members then copy and apply these operations in an asynchronous manner.

All members maintain a copy of the oplog.

### OPLOG

*Oplog* is short form for operation log. It's a capped collection which keeps rolling record of all the operations that modifies the data stored in the database.

*The oplog is stored in a special database called local, in the oplog.\$main collection. Each document in the oplog represents a single operation performed on the master server. The document contains several keys, including the following:*

**ts:** Timestamp for the operation. The timestamp type is an internal type used to track when operations are performed. It is composed of a 4-byte timestamp and a 4-byte incrementing counter.

**op:** Type of operation performed as a 1-byte code (e.g., "i" for an insert).

**ns:** Namespace (collection name) where the operation was performed.

**o:** Document further specifying the operation to perform. For an insert, this would be the document to insert.

*Oplog stores details of only the operations that changes the data for e.g. a query will not be stored in the oplog, as oplog is a mechanism for keeping the data on the secondary nodes in sync with the primary.*

*The operations stored in the oplog are not exactly those that were performed on the master server itself, they are transformed before being stored so that they remain idempotent which means that it can be applied any number of times on the secondary node but the result will be consistent.*

*Since oplog is a capped collection hence as new operations are appended into the collection it will automatically start moving out the oldest operations, this is done to ensure that it does not grow beyond a pre-set bound which is the oplog size.*

**Oplog size** – MongoDB creates an oplog of a default size when the replica set member first starts up which depends on the OS.

*By default, for 64-bit Linux and Windows instances MongoDB will use 5% of available free space for the oplog, if this is lower than a*

*Gigabyte then MongoDB allocates 1 GB of space.*

*Though the default size is sufficient in most of the cases, but if the user wants they can use –oplogsize option when starting the server to specify the size of oplog in Megabytes.*

*This space will be allocated in the local database and will be pre-allocated when the server starts.*

*If the user has the following workload then there might be a requirement of reconsidering the oplog size:*

- **Updates to multiple documents simultaneously** – since the operations need to be translated into operations which are idempotent hence this scenario might end up requiring great deal of oplog size.
- **Deletion and Insertions happening at the same rate involving same amount of data** – In this scenario though the database size will not increase but the operations translation into idempotent operation can lead to oplog that can be quite large.
- **Significant number of in-place updates** – Though these updates will not change the database size but the recording of updates as idempotent operations in the oplog can lead to a bigger oplog.

## **Initial sync and Replication**

Initial sync is done when the member is in either of the following two cases

1. The node has started for the first time i.e. it's a new node and has no data.
2. Or the node has become stale where the primary has overwritten the oplog and the node has not replicated the data. In this case the data will be removed.

In both the cases the initial sync involves the following steps

1. First all the databases are cloned.
2. Using oplog of the source node the changes are applied to its dataset

3. And finally the indexes are built on all the collections.

Post the initial sync the replica set members continuously replicates the changes in order to be up-to-date.

 *Mostly the synchronization happens from the primary however chained replication can be enabled where the sync happens from a secondary only i.e. the sync targets are changed based on the ping time and state of other member's replication.*

### **Syncing – Normal Operation**

When a secondary is operating normally, it first chooses a member to sync from and then starts pulling operations from the source's local.oplog.rs collection.

When it gets an operation say for e.g. W for write in this case, it does the following steps:

1. Applies the op
2. Writes the op to its own oplog (also local.oplog.rs)
3. Requests for the next op

**Scenario:** Let's suppose it crashes between step 1 & 2 and then again comes back. In this scenario it'll assume the operation has not been performed and will re-apply it.

Since oplog ops are idempotent hence the same operation which in this case is W can be applied once, twice, or a thousand times and then also we'll end up with the same document.

For example, if we have a doc that looks like

{l:1}

And an increment operation is performed on the same like

{\$inc:{l:1}} on the primary

In this case the primary oplog will store {l:2}.

The secondaries will replicate that instead of the \$inc. Hence the value remains the same even if the log is applied multiple times.

### Starting up

When you start up a node, it takes a look at its local.oplog.rs collection and finds the latest entry. This is called the lastOpTimeWritten and it's the latest op this secondary has applied.

*The following shell helper can be used to get the current last op in the shell:*

`> rs.debug.getLastOpWritten()`

*The "ts" field is the last optime written.*

If a member starts up and there are no entries in the oplog, it will begin the initial sync process. If it has the last op time then it will choose a target to sync from and sync the data with it as syncing under normal operations.

## Who to sync from?

In this section we will look at how the source is chosen to sync from.

As of 2.0, servers automatically sync from “nearest” node based on average ping time.

So, if you bring up a new member it starts sending out heartbeats to all the other members and monitors how long it takes to get a response.

Once it has a decent picture of the world, it'll decide who to sync from using the following algorithm:

```
for each member that is healthy:  
    if member[state] == PRIMARY  
        add to set of possible sync targets  
        if member[lastOpTimeWritten] >  
            our[lastOpTimeWritten]  
            add to set of possible sync targets  
        sync target = (member with the min ping time  
        from the possible sync targets)
```

 The definition of “member that is healthy” has changed somewhat over the versions, but generally you can think of it as a “normal” member: a primary or secondary.

*In 2.0, “healthy” debatably includes slave delayed nodes.*

*Running the following command will show the server which is chosen as source for syncing.*

`db.adminCommand({replSetGetStatus:1})`

*The output field “syncingTo” is present only on secondary nodes and provides information on the node from which it is syncing..*

## Making writes work with Chaining Slaves

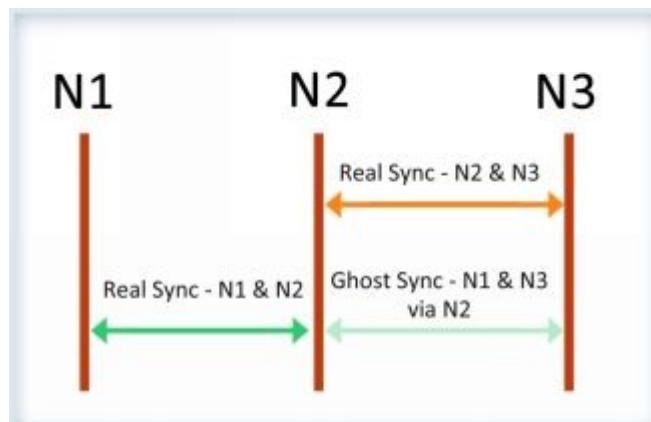
💡 We have seen above that the algorithm for choosing a sync source means that slave chaining is semi-automatic: start up a server in a data center and it'll (probably) sync from a server in the same data center, minimizing WAN traffic.

*Note this will never lead to a sync loop i.e. N1 syncing from N2 and N2 syncing from N1. Reason being a secondary can only sync from another secondary with a strictly higher optime*

In this section we will see how w (write operation) works with slave chaining. Say for e.g. if N1 is syncing from N2 which is further syncing from N3 then in this case how N3 will know that till which point N1 is synced to. When N1 starts syncing from N2, it sends a special “handshake” message which basically says, “Hi, I'm N1 and I'll be syncing from your oplog. Please track this connection”.

When N2 gets this message, it says, “Hmm, I'm not primary, so let me forward that along to the member I'm syncing from.” So it opens a new connection to N3 and says “Pretend I'm 'N1', I'll be syncing from your oplog on N1's behalf.” Note that N2 now has two connections open to N3, one for itself and one for N1.

Whenever N1 requests more ops from N2, N2 sends the ops from its oplog and then forwards a dummy request to N3 along “N1's” connection to N3. N1 doesn't even need to be able to connect directly to N3.



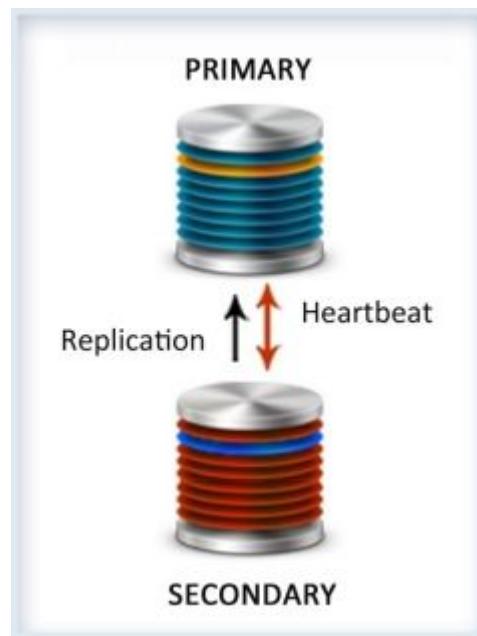
**Fig 7-4: Writes via chaining Slaves**

Though this minimizes network traffic on a positive side however on the negative side the absolute time it takes a write to get to all members is higher.

## Failover

In this section we will look at how primary and secondary member failovers are handled in Replica sets.

All the members within the replica set are interconnected to each other and they exchange heartbeat message amongst each other.



**Fig 7-5: Heartbeat message exchange**

Hence a node with missing heartbeat will be considered as crashed.

### If Node is Secondary Node

If the node is a secondary node, then it will be removed from the replica set membership. After it recovers in future it can re-join the cluster, now it needs to update the data with the latest changes.

1. If the down period is small then it connects to the primary and catch up with the latest updates.
2. However if the down period is lengthy then the secondary server will need to resync with primary where it deletes all

its data and do initial sync as if it's a new server.

## If Node is Primary Node

If the node is a primary node and majority of the original set members can still connect to each other than these nodes will elect a new primary which is in accordance to the automatic failover capability of the replica set.

This election process will be initiated by any node that cannot reach the primary.

The new primary must be elected by a majority of the nodes in the set. Arbiter nodes are useful for breaking ties (e.g., when the participating nodes are split into two halves separated by a network partition).

The new primary will be the node with the highest priority, using freshness of data to break ties between nodes with the same priority.

The primary node uses a heartbeat to track how many nodes in the cluster are visible to it. If this falls below a majority, the primary will automatically fall back to secondary status. This prevents the primary from continuing to function as such when it is separated from the cluster by a network partition.

## Rollbacks

Whenever the primary changes, the data on the new primary is assumed to be the most up-to-date data in the system. Any operations that have been applied on any other nodes (i.e., the former primary node) will be rolled back, even if the former primary comes back online.

In this scenario when the failed primary joins back the set then the operations which were only applied on it are rolled back first before syncing with the new primary. The rollback operation reverts all the write operations that were not replicated across replica set. This is done in order to maintain database consistency across replica set.

To accomplish this rollback, all nodes go through a resync process when connecting to a new primary. They look through their oplog for

operations that have not been applied on the primary and query the new primary to get an up-to-date copy of any documents affected by such operations. Nodes that are currently in the process of resyncing are said to be recovering and will not be eligible for primary election until the process is complete.

This happens very rarely and if it happens it is often the result of a network partition with replication lag where the secondaries cannot keep up with the throughput of operations on the former primary.

*It needs to be noted that a rollback does not occur if the write operations replicate to another member of the replica set before the primary steps down and if that member remains available and accessible to a majority of the replica set.*

The rollback data is written to a [BSON](#) file with filenames such as <database>. <collection>. <timestamp>. bson in the database's [dbpath](#) directory.

The administrator can decide to apply or ignore the rollback data. The applying of rollback data can only begin when all the nodes are in sync with the new Primary and have rolled back to a consistent state.

Bsondump can be used to read the contents of these rollback files which then need to be manually applied to the new primary using mongorestore.

There is no way for MongoDB to appropriately and fairly handle rollback situations automatically.

Therefore manual intervention is required to apply rollback data. While applying the rollback it's important to ensure that these are replicated to either all or at least some of the members in the set so that in case of any failover rollbacks can be avoided.

## Consistency

We have seen that the replica set members keep on replicating data among each other by reading the oplog. Next question which arises is how the consistency of data is maintained. In this section we will

look at how MongoDB ensures that the user always accesses consistent data.

*In MongoDB though the reads can be routed to the secondaries but the writes are always routed to the primary hence eradicating the scenario where two nodes are simultaneously trying to update the same data set. Hence the data set on the primary node is always consistent.*

If the user's read requests are routed to the primary node he will always see up-to-date changes which mean the read operations are always consistent with the last write operations.

However if the application has changed the read preference to read from secondaries then there might be a probability of user not seeing the latest changes or seeing previous states, this is because the writes are replicated asynchronously on the secondaries.

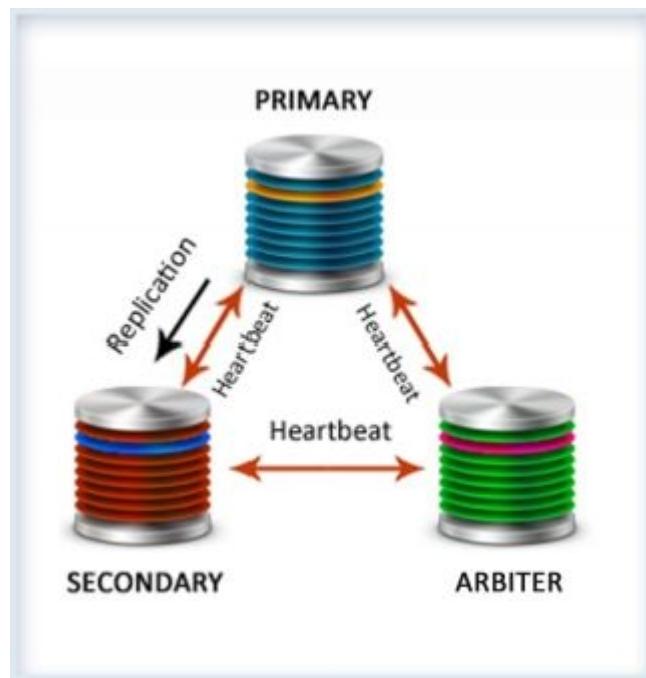
 *This behavior is sometimes characterized as eventual consistency which implies that though the secondary's state is not consistent with the primary node state but will eventually become consistent over time.*

There is no way to guarantee consistency for reads from secondary members, except by configuring the [client driver](#) to ensure that write operations succeed on all members before completing successfully by issuing write concerns which we will be discussing in a while.

## Possible Replication deployment

The architecture you chose for deploying a replica set affects its capacity and capability. In this section we will look at few strategies which we need to be aware of while deciding on the architecture and will also be discussing the deployment architecture.

1. The replica set should always have odd number of members. This should be done in order to ensure that there is no tie while electing primary. If number of nodes is even then arbiter can be used as to ensure that the total nodes participating in election is odd.



*Fig 7-6: Members Replica Set with Primary, Secondary and Arbiter*

2. Fault tolerance of a replica set is the number of members that can become unavailable and still leave enough members in the set to elect a primary. The following table indicates the relationship between the number of members in the set and its fault tolerance. Hence the fault tolerance should be considered while deciding on the number of members

No. Of Members	Majority required for electing a primary	Fault Tolerance
----------------	--	-----------------

**3**

**2**

**1**

**4**

**3**

**1**

**5**

**3**

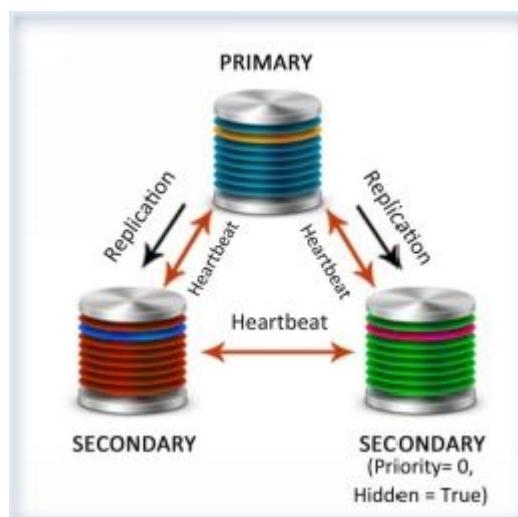
**2**

**6**

4

2

3. If the application has specific dedicated requirements such as for backups or reporting then hidden or delayed members can be considered as part of the replica set.



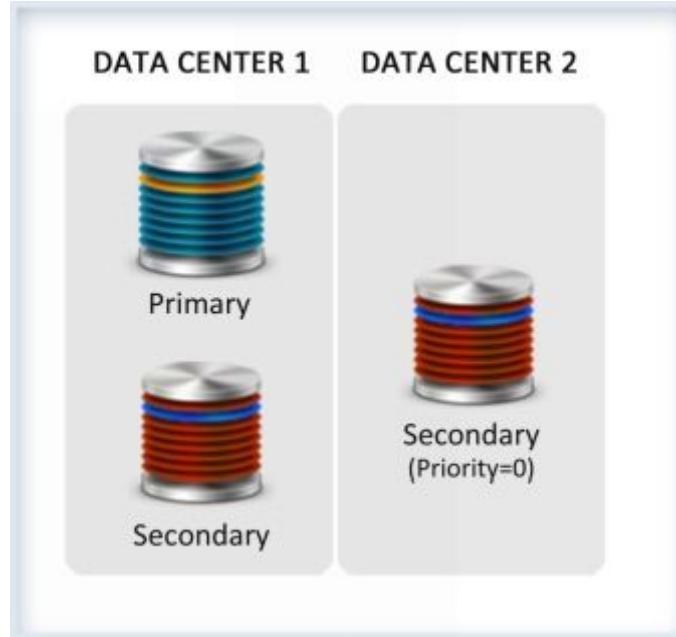
**Fig 7-7: Members Replica Set with Primary, Secondary and Hidden member**

4. If the application is read heavy then in that case the read can be distributed across secondaries.

As the requirement increases more nodes can be added for increasing the data duplication and hence can have a positive impact on the read throughput.

5. The members should be distributed geographically in order to cater to main data center failure.

The members which are kept at a geographically different location other than the main data center can have priority set as 0 so that they cannot be elected as primary and hence act as a standby only.



**Fig 7-8: Members Replica Set with Primary, Secondary and a Priority 0 member distributed across data center**

6. When a replica set has members in multiple data centers, network partitions can prevent communication between data centers. In an election, members must see each other to create a majority. To ensure that the replica set members can confirm a majority and elect a primary, keep a majority of the set's members in one location.

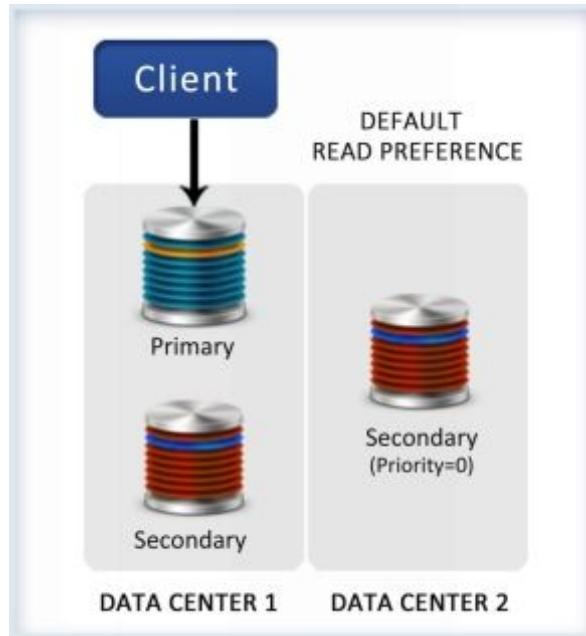
## Scaling Reads

Though the primary purpose of the secondaries is to ensure data availability in case of downtime of the primary node, however there are other valid use cases also for secondaries such as it can be used dedicatedly for performing backup operations or it be used for scaling out reads or for performing data processing jobs. One way to scale reads with MongoDB is to issue queries against the secondary nodes, by doing so the workload on the master is reduced.

*One important point that we need to consider when using secondaries to scale reads in MongoDB is that the replication is asynchronous. This means that when data is inserted or updated on the master, the data on the secondary will be out-of-date momentarily.*

Hence if the application in question is read heavy and is accessed over network and does not need an up-to-date data then the secondaries can be used to scale out the read in order to provide a good read throughput.

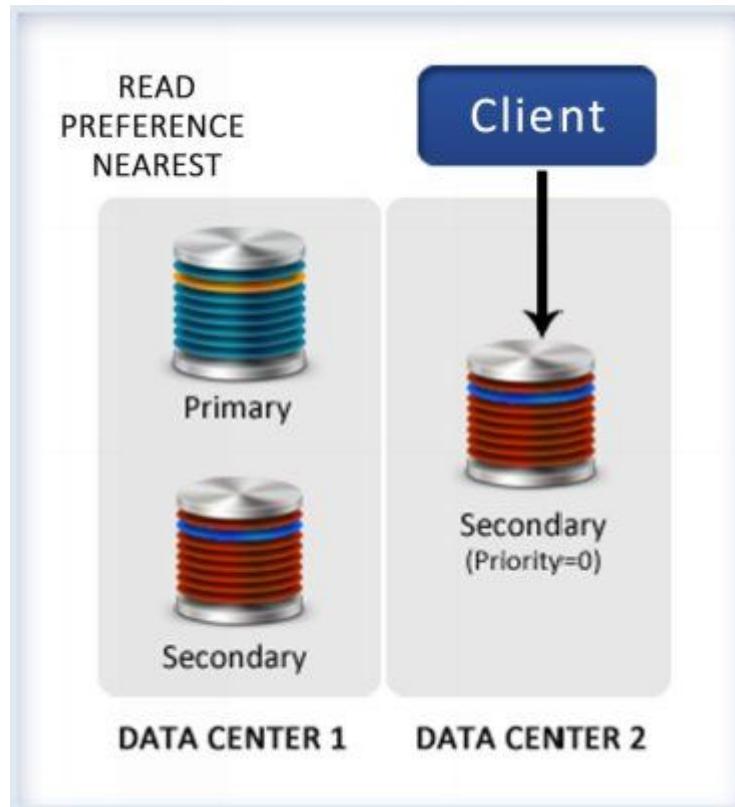
*Though by default the read requests are routed to the primary node but the requests can be distributed over secondary nodes by specifying the **read preferences**.*



*Fig 7-9: Default Read Preference*

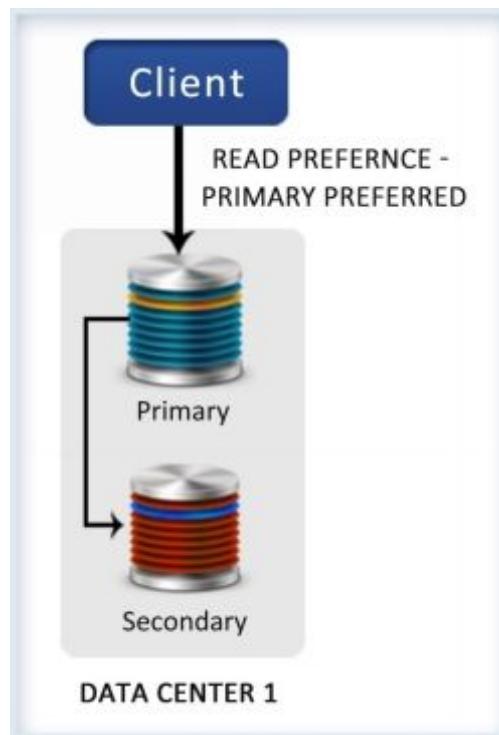
The following are ideal use cases where by routing the reads on secondary node can help gain a significant improvement in the read throughput and also can help reduce the latency.

1. Applications which are geographically distributed. In such cases we can have a replica set which is distributed across geographies and the read preferences should be set to read from the **nearest** secondary node. This helps in reducing the latency which is caused when reading over network and hence improves the read performance.



***Fig 7-10: Read Preference – Nearest***

2. If the application always requires an up to date data then use the option primaryPreferred, which in normal circumstances will always read from the primary node but in case of emergency it will route the read to secondaries. This is useful during failovers.



***Fig 7-11: Read Preference – primaryPreferred***

3. If we have an application that supports two types of operations, the first operation is the main workload which involves reading and doing some processing on the data whereas the second operation is generating reports using the data, then in such scenarios we can have the reporting reads directed to the secondaries.

MongoDB supports the following read preference modes.

1. primary – This is the default mode, all the read requests are routed to the primary node.
2. primaryPreferred – In normal circumstances the reads will be from primary but under emergency such as primary not available reads will be from the secondary nodes.
3. Secondary – Reads from the secondary members
4. secondaryPreferred – reads from secondary members and if secondaries are unavailable then read from the primary
5. nearest – reads from the nearest member of the replica set.

*In addition to scaling reads as we mentioned in the second ideal use case for using secondaries is to offload intensive processing, aggregating, and administration tasks on the secondaries in order to avoid degrading the primary's performance.*

*Hence we can perform blocking operations on the secondary without ever affecting the performance of the primary node.*

## Application Write Concerns

When the client application is interacting with MongoDB they are generally not aware whether the database is on standalone deployment or is deployed as a replica set.

However when dealing with replica sets the client should be aware about **write concern** and **read concern**.

Since replica set duplicates the data and stores it across multiple nodes, these two concerns give client application the flexibility to enforce data consistency across nodes while performing read or write operations.

*MongoDB is safe off by default that is it doesn't wait for any response of success or failure from the server when issuing a write operation. However in certain applications it might be required to at least receive confirmation that the write is successful.*

*Using Write concern enables the application in getting a success or failure response from MongoDB.*

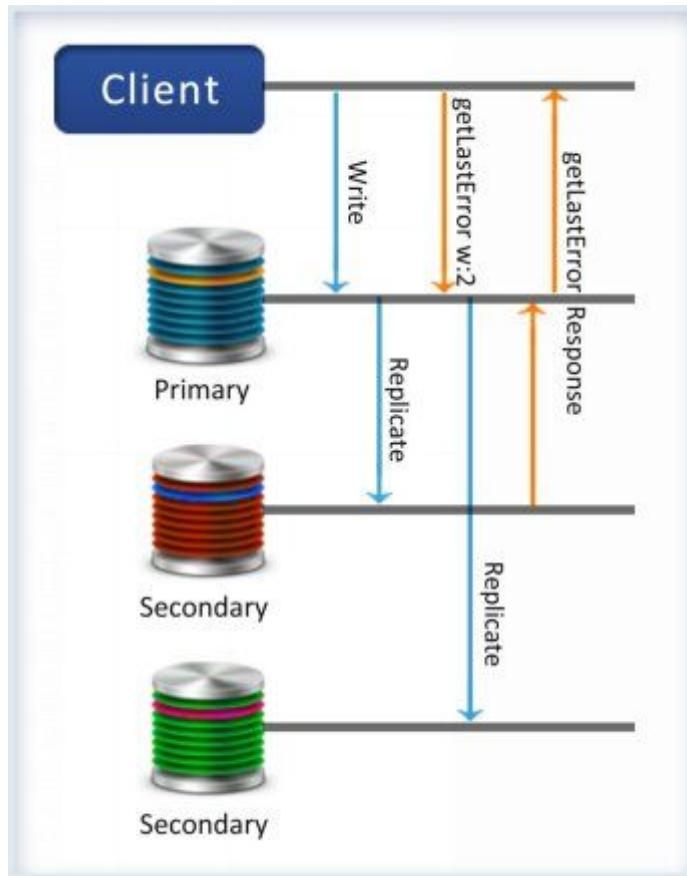
When used in a replica set deployment of MongoDB, the write concern sends a confirmation from the server to the application that the write has succeeded on the primary node. However this can be configured so that the write concern returns success only when the write is replicated to all the nodes maintaining the data.

*In practical scenario this won't be feasible as it will reduce the write performance hence ideally the client can ensure using write concern that the data is replicated to one more node in addition to primary so that the data is not lost even if the primary steps down.*

Write concern uses the `getLastError` command after write operations to return an object with error information or confirmation that there are no errors.

**getLastError** command with `w` option ensures that the write has replicated to the specified number of members. Either a number or majority can be specified as value of `w` option.

If number is specified then the write replicates to that many number of nodes before returning success whereas if majority is specified then the write is replicated to majority of members before returning the result.



**Fig 7-12: getLastError**

 If while specifying number, the number is greater than the nodes which actually hold the data then in that case the command will keep on waiting until the members are available. In order to avoid this indefinite wait time, **wtimeout** should also be used along with **w** which will ensure that it will wait for the specified time period and if the write has not succeeded by that time it will time out.

## How writes happen with write concern

In order to ensure that the written data is present on say at least two members the result can be achieved by issuing the following command

```
> db.foo.runCommand({getLastError:1, w:2})
```

In order to understand how this command will be executed let's say we have a member called primary and another member namely secondary syncing from it.

Before we look at the steps the question that arises is how does the primary know where secondary is synced to? Since secondary is querying primary's oplog for more op results to be applied hence if the secondary requests an op written at say  $t$  time, it implies to the primary that secondary has replicated all ops written before  $t$ .

Hence the steps that a write concern follows are:

1. The write operation is directed to the primary.
2. The Write is written to the oplog on primary, with a field "ts" saying the write occurred at time  $t$ .
3. Since {getLastError: 1, w: 2} is called on primary and primary has completed the write operation, hence it is just waiting for one more server to get the write (reason w is specified as 2).
4. secondary queries the oplog on primary and gets the op
5. secondary applies the op from time  $t$
6. Next secondary requests ops with {ts: {\$gt: t}} from primary's oplog
7. primary updates that secondary has applied up to  $t$  because it is requesting ops  $> t$ .
8. getLastError notices that primary and secondary both have the write, hence w:2 is satisfied and hence it returns success.

## ***Implementing Advanced Clustering with Replica Sets***

Having learnt the architecture and inner workings of replica sets, we will now focus on administration and usage of replica sets.

We will be focusing on the following:

1. *Get the replica set member up and running*
2. *Remove server from the replica set*
3. *Add a server to a replica set.*
4. *Add an arbiter to the replica set.*
5. *Inspect the status.*
6. *Force a new election of primary*
7. *Use the web interface to inspect the status of your replica set.*

*The following examples assume a replica set named testset that has the following configuration. It has three members – two active and one passive.*

Service	Daemon	Address	Dbpath
<b>Active Member 1</b>	Mongod	[hostname]:27021	C:\db\active1\data
<b>Active Member 2</b>	Mongod	[hostname]:27022	C:\db\active2\data
<b>Passive Member 1</b>	Mongod	[hostname]:27023	C:\db\passive1\data

*The hostname used in the above table can be found out using the following command.*



`C:\>hostname`

ANOC9

C:\>

*In the examples that follow, substitute the [hostname] highlighted in red with whatever value is returned by running the hostname command on your own system.*

## Getting a Replica Set Member Up and Running

In order to get the replica set up and running we need to make all the active members up and running.

First step is to get the first active member up and running.

Open a terminal window and create the data directory

```
C:\>mkdir C:\db\active1\data
```

C:\>

Connect to the mongod

```
c:\mongodb\bin>mongod --dbpath C:\db\active1\data -  
-port 27021 --replSet testset/ANOC9:27022 --rest
```

```
Tue Feb 11 19:41:16.293 [initandlisten] MongoDB  
starting : pid=4964 port=27021
```

```
.....  
Tue Feb 11 19:41:30.628 [rsStart] replSet can't get  
local.system.replset config  
from self or any seed (yet)
```

The --replSet option tells the instance the name of the replica set it is joining, as well as the name of at least one other member of the set.

This is the first member of the replica set, so you can give it the address of any other member, even if that member has not been started yet.

Only one member address is required, but you can also provide the names of other members by separating their addresses with commas, as shown in the following example:

```
mongod --dbpath C:\db\active1\data -port 27021  
-replSet testset/[hostname]:27022,  
[hostname]:27023 --rest
```

In order to keep things simple, in this example we have used only one address.

Next step is to get the 2<sup>nd</sup> Active member up and running.

Open a terminal window and create the data directory for the second active member.

```
C:\>mkdir C:\db\active2\data  
C:\>
```

Connect to mongod as below

```
c:\mongodb\bin>mongod --dbpath C:\db\active2\data -  
-port 27022 --repSet testset/ANOC9:27021 --rest  
Tue Feb 11 19:50:43.735 [initandlisten] MongoDB  
starting : pid=4144 port=27022 d  
bpath=C:\db\active2\data 64-bit host=.....  
rs.initiate() in the shell -- if that is not already done
```

Finally we need to get the passive member up and running.

Open a separate window and create the data directory for the passive member.

```
C:\>mkdir C:\db\passive1\data  
C:\>
```

Connect to mongod

```
c:\mongodb\bin>mongod --dbpath C:\db\passive1\data  
--port 27023 --repSet testset/ANOC9:27021 --rest  
Tue Feb 11 19:55:52.530 [initandlisten] MongoDB  
starting : pid=4604 port=27023 d  
bpath=C:\db\passive1\data 64-bit host=.....  
Tue Feb 11 19:55:52.768 [rsStart] repSet info you may  
need to run repSetInitia  
te -- rs.initiate() in the shell -- if that is not already done
```

*The --rest option in the preceding example activates a REST interface on port+1000 (i.e., for port 27021, the REST interface is available on port 28021). This approach enables you to use the web interface to inspect the status of your replica set.*

At this point, we have three server instances up and running and communicating with each other; however, the replica set is not initialized hence in the next step we will initialize the replica set and instruct each member about its role and responsibilities.

In order to initialize the replica set we will be connecting to one of the servers. In this example it is the first server which is running on port 27021.

Open a new command prompt and connect to the mongo interface for the first server

```
C:\>cd c:\mongodb\bin
```

```
c:\mongodb\bin>mongo ANOC9 --port 27021
```

```
MongoDB shell version: 2.4.9
```

```
connecting to: 127.0.0.1:27021/ANOC9
```

```
>
```

Switch to admin database

```
> use admin
```

```
switched to db admin
```

```
>
```

Next set up a configuration data structure that lists all the servers and their respective roles:

```
> cfg = {
... _id: 'testset',
... members: [
... {_id:0, host: 'ANOC9:27021'},
... {_id:1, host: 'ANOC9:27022'},
... {_id:2, host: 'ANOC9:27023', priority:0}
... ]
... }
{ "_id" : "testset",
  "members" : [
    {
      "host": "ANOC9:27021",
      "priority": 1,
      "tags": {
        "type": "primary"
      }
    },
    {
      "host": "ANOC9:27022",
      "priority": 2,
      "tags": {
        "type": "secondary"
      }
    },
    {
      "host": "ANOC9:27023",
      "priority": 3,
      "tags": {
        "type": "secondary"
      }
    }
  ]
}
```

```
"_id" : 0,  
"host" : "ANOC9:27021"  
,  
.....  
{  
"_id" : 2,  
"host" : "ANOC9:27023",  
"priority" : 0  
} ]}>
```

The structure of the replica set is configured.

*Notice the use of the “priority 0” in the configuration structure to specify that the passive member is not to be considered as a possible candidate for promotion to the primary role.*

In the next step we will issue the command to initialize the replica set.

```
> rs.initiate(cfg)  
{  
  "info" : "Config now saved locally. Should come online  
in about a minut  
e.",  
  "ok" : 1  
}  
>
```

Let's now check the status of the replica set to determine whether it has been set up correctly:

```
testset:PRIMARY> rs.status()  
{  
  "set" : "testset",  
  "date" : ISODate("2014-02-11T14:41:13Z"),  
  "myState" : 1,  
  "members" : [
```

```
{  
  "_id" : 0,  
  .....  
  "ok" : 1  
}  
testset:PRIMARY>
```

The output in the preceding example indicates that all is OK. The replica set is now successfully configured and initialized.

*Let's see how we can determine the primary node. In order to do so connect to any of the member of the replica set and issue the following command and verify the primary.*

```
testset:PRIMARY> db.isMaster()
{ "setName" : "testset",
  "ismaster" : true,
  "secondary" : false,
  .....
  "primary" : "ANOC9:27021",
  "me" : "ANOC9:27021",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2014-02-11T14:47:42.170Z"),
  "ok" : 1
}testset:PRIMARY>
```

## Removing a Server from Replica Set

In this example we will be removing the secondary active member from the set.

Let's connect to the secondary member mongo instance.

Open a new command prompt as shown below:

```
C:\>cd c:\mongodb\bin
```

```
c:\mongodb\bin>mongo ANOC9 --port 27022
MongoDB shell version: 2.4.9
connecting to: 127.0.0.1:27022/ANOC9
testset:SECONDARY>
```

Issue the following command in the mongo shell to shut down the instance:

```
testset:SECONDARY> use admin
switched to db admin
testset:SECONDARY> db.shutdownServer()
```

*Tue Feb 11 20:16:19.939 DBClientCursor::init call() failed  
server should be down.....*

>

Next we need to connect to the primary member mongo console, switch to the admin database using “use admin” command and issue the following to remove the member from the replica set

*testset:PRIMARY> use admin*

*switched to db admin*

*testset:PRIMARY> rs.remove("ANOC9:27022")*

*Tue Feb 11 20:20:38.579 DBClientCursor::init call() failed*

*Tue Feb 11 20:20:38.588 JavaScript execution failed:  
Error: error doing query: f*

*ailed at src/mongo/shell/query.js:L78*

*Tue Feb 11 20:20:38.589 trying reconnect to  
127.0.0.1:27021*

*Tue Feb 11 20:20:38.639 reconnect 127.0.0.1:27021 ok*

*testset:PRIMARY>*

In order to vet whether the member is removed or not we can issue the *rs.status()* command

## Adding a Server to a Replica Set

We have a replica set up and running, next we will add a new active member to the set.

As with other members we will begin by opening a new command prompt and creating the data directory first.

*C:\>mkdir C:\db\active3\data*

*C:\>*

Next we will start the mongod using the following command.

*c:\mongodb\bin>mongod --dbpath C:\db\active3\data -  
-port 27024 --repSet testset/ANOC9:27021 --rest*

*.....*

We have the new mongod running, now we need to add this to the replica set. For this we will connect to the primary's mongo console.

```
C:\>c:\mongodb\bin\mongo.exe --port 27021
```

```
MongoDB shell version: 2.4.9  
connecting to: 127.0.0.1:27021/test  
testset:PRIMARY>
```

Next we will switch to admin db.

```
testset:PRIMARY> use admin  
switched to db admin  
testset:PRIMARY>
```

Finally we need to issue the following command to add the new mongod to the replica set.

```
testset:PRIMARY> rs.add("ANOC9:27024")  
{ "ok" : 1 }
```

The replica set status can be checked to vet whether the new active member is added or not using `rs.status()`

## Adding an Arbiter to a Replica Set

In this example we will add an arbiter member to the replica set.

As with the other members we will begin by creating the data directory for the MongoDB instance.

```
C:\>mkdir c:\db\arbiter\data  
C:\>
```

We will next start the mongod using the following command

```
c:\mongodb\bin>mongod --dbpath c:\db\arbiter\data --  
port 30000 --repSet testset/ANOC9:27021 --rest  
Wed Feb 12 12:51:46.378 [initandlisten] MongoDB  
starting : pid=5480 port=30000 d  
bpath=c:\db\arbiter\data 64-bit .....  
from self or any seed (yet)
```

Connect to the primary's mongo console, switch to the admin db and add the newly created mongod as an arbiter to the replica set.

```
C:\>c:\mongodb\bin\mongo.exe --port 27021
```

```
MongoDB shell version: 2.4.9
connecting to: 127.0.0.1:27021/test
testset:PRIMARY> use admin
switched to db admin
```

```
testset:PRIMARY> rs.addArb("ANOC9:30000")
{ "ok" : 1 }
testset:PRIMARY>
```

Whether the step is successful or not the same can be verified using `rs.status()`

## Inspecting an Instance's Status with `rs.status()`

We have been referring to `rs.status()` throughout our examples above to check the status of replica set. In this section we will briefly understand what this command is all about. The `rs.status()` command is probably the most common command that is being used when working with replica sets.

It allows the user to check the status of the member whose console they are connected to and also enables them to view its role within the replica set.

The following command is issued from the primary's mongo console.

```
testset:PRIMARY> rs.status()
{
  "set" : "testset",
  "date" : ISODate("2014-02-12T07:26:33Z"),
  "myState" : 1,
  "members" : [
    .....
    "ok" : 1
}
```

*testset:PRIMARY>*

*The **myState** field's value indicates the status of the member and it can have the following values:*

<b>myState</b>	<b>Description</b>
----------------	--------------------

**0**

*Member is starting up and is in phase 1.*

**1**

*Member is operating as a primary (master) server.*

**2**

*Member is operating as a secondary server*

**3**

*Member is recovering; the sysadmin has restarted the member server in recovery mode after a possible crash or other data issue*

**4**

*Member has encountered a fatal error; the errMsg field in the members array for this server should show more details about the problem.*

**5**

*Member is starting up and has reached phase 2.*

**6**

*Member is in an unknown state; this could indicate a misconfigured replica set, where some servers are not reachable by all other members.*

7

*Member is operating as an arbiter*

**8**

*Member is down or otherwise unreachable. The lastheartbeat timestamp in the member array associated with this server should provide the date/time that the server was last seen alive.*

*In the preceding command results the information returned for this command shows that the primary server is operating with a myState value of 1 which depicts that the “Member is operating as a primary (master).”*

## Forcing a New Election with rs.stepDown()

The rs.stepDown () command can be used to force a primary server to step down and also forces the election of a new primary server.

*This command is useful in the following situations:*

1. You would like to take the server hosting the primary instance offline, whether to investigate the server or to implement hardware upgrades or maintenance.
2. You would like to run a diagnostic process against the data structures.
3. You would like to simulate the effect of a primary failure and force your cluster to fail over, enabling you to test how your application responds to such an event.

The following shows the output of the command when run against the testset replica set.

```
testset:PRIMARY> rs.stepDown()
```

```
Wed Feb 12 13:12:50.640 DBClientCursor::init call() failed
```

```
Wed Feb 12 13:12:50.655 JavaScript execution failed:  
Error: error doing query: failed at src/mongo/shell/query.js:L78
```

```
Wed Feb 12 13:12:50.656 trying reconnect to 127.0.0.1:27021
```

```
Wed Feb 12 13:12:50.658 reconnect 127.0.0.1:27021 ok  
testset:SECONDARY>
```

*After execution of the command the prompt changed from testset:PRIMARY to testset:SECONDARY.*

*rs.status() can be used to check whether the stepDown () is successful or not.*

*Please note the myState value it returns is 2 now, which means the “Member is operating as secondary.”*

## Viewing Replica Set Status with the Web Interface

MongoDB maintains a web-based console for viewing the status of the system. For our example we can access the console by opening the URL `http://localhost:28021` with your web browser.



**Fig 7-13: Web Interface**

*The port number of the web interface is set by default to port  $n+1000$ , where  $n$  is the port number of your instance. Hence assuming our primary instance is on port 27021, as in this chapter's example, its web interface can be found on port 28021.*

If we open this interface in our web browser, we will see a link to the status of the replica set at the top of the page.

Clicking the Replica set status link will take you to the Replica Set dashboard as shown in the Figure

The screenshot shows a Microsoft Internet Explorer window with the title "Replica Set Status ANOC9:27021 - Windows Internet Explorer". The address bar shows the URL "http://localhost:28021/\_repSet". The main content area has a header "Home | View Repiset Config | repSetGetStatus | Docs". Below this, it says "Set name: testset", "Majority up: yes", and "Lag: 0 secs". A table titled "Members" lists four members:

Member	Id	Up	cctime	Last heartbeat	Votes	Priority	State	Messages	optime	clock skew
ANOC9:27021 (me)	0	1	44 mins		1	1	SECONDARY	syncing to: ANOC9:27024	52b21e5.1	
ANOC9:27023	2	1	49 mins	2 secs ago	1	0	SECONDARY		52b21e5.1	
ANOC9:27024	3	1	33 mins	2 secs ago	1	1	PRIMARY		52b21e5.1	
ANOC9:30000	4	1	24 mins	2 secs ago	1	1	ARBITER		0.0	

Below the table, there is a section titled "Recent repiset log activity" with the following log entries:

```
Wed Feb 12 12:35:05.193 [reStart] trying to connect ANOC9:27022
12:35:06.187 [reStart] couldntdeposit connect to ANOC9:27022; couldntdeposit connect to server ANOC9:27022
12:35:07.181 [reStart] repiset I am ANOC9:27021
12:35:07.182 [reStart] repiset warning command line need ANOC9:27022 is not present in the current repiset config
12:35:07.191 [reStart] repiset STARTUP2
12:35:07.192 [reStart] repiset total number of voters is even - add arbiter or give one member an extra vote
12:35:08.205 [reSync] repiset SECONDARY
12:35:10.178 [reHealthPoll] couldntdeposit connect to ANOC9:27023; couldntdeposit connect to server ANOC9:27023
12:35:11.178 .
12:35:12.170 [reHealthPoll] repiset info ANOC9:27023 heartbeat failed, retrying
```

**Fig 7-14: Replica set status report**

# Sharding

We have seen in the previous section how replica set in MongoDB is used to duplicate the data in order to save against any adversity and distribute the read load in order to increase the read efficiency.

*MongoDB makes extensive use of RAM for low latency database operations. In MongoDB, all data is read and manipulated through memory-mapped files. Reading data from memory is measured in nanoseconds and reading data from disk is measured in milliseconds; and so reading from memory is approximately 100,000 times faster than reading from disk.*

*The set of data and indexes that are accessed most frequently during normal operations is called the working set, which ideally should fit in RAM. It may be the case that the working set represents a fraction of the entire database.*



*Page faults occur when MongoDB attempts to access data that has not been loaded in RAM. If there is free memory then the operating system will locate the page on disk and load it into memory directly. However, if there is no free memory the operating system must write a page that is in memory to disk and then read the requested page into memory. This process will be slower than accessing data that is already in memory. Some operations may inadvertently purge a large percentage of the working set from memory, which adversely affects performance.*

*For example, a query that scans all documents in the database, where the database is larger than the RAM on the server, will cause documents to be read into memory and the working set to be written out to disk.*

*Ensuring you have defined appropriate index coverage for your queries during the schema design phase of the project will minimize the risk of this happening. The MongoDB explain operation can be used to provide information on your query plan and indexes used.*

*A useful output included with MongoDB's serverStatus command is a workingSet document that provides an estimated size of the MongoDB instance's working set. Operations teams can track the number of pages accessed by the instance over a given period, and the elapsed time from the oldest to newest document in the working set. By tracking these metrics, it is possible to detect when the working set is approaching current RAM limits and proactively take action to ensure the system is scaled.*

In MongoDB the scaling is handled by scaling out the data horizontally i.e. partitioning the data across multiple commodity servers, which is also termed as Sharding (Horizontal Scaling).

Sharding addresses the challenge of scaling to support high throughput and large data sets by horizontally dividing the datasets across servers where each server is responsible for handling its part of data and no one server is burdened. These servers are also termed as shards.

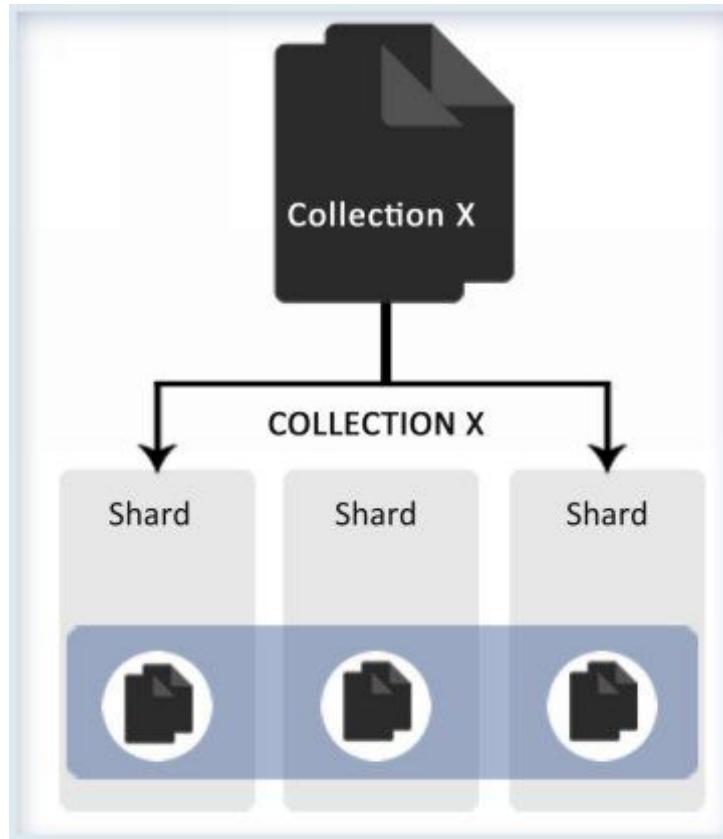
Each shard is an independent database, and collectively, the shards make up a single logical database.

*Sharding reduces the number of operations each shard handles. For e.g. to insert data the application only need to access the shard which is responsible for storing that records.*

*The processes that need to be handled by each shard reduce as the cluster grows because the subset of data that the shard holds reduces. As a result, shared clusters can increase capacity and throughput horizontally.*

*Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.*

*For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data. The following diagram depicts how a collection which is sharded will appear when distributed across 3 shards.*



**Fig 7-15: Sharded collection across 3 shards**

While Sharding is a powerful and compelling feature, a sharded cluster has significant infrastructure requirements and increases the overall complexity of a deployment. Hence we need to understand the scenarios where we might consider using Sharding as needlessly using Sharding will only add complexities to the system without reaping much benefit.

*Use Sharding if:*

1. *The size of the dataset is huge and it has started challenging the capacity of a single system.*
2. *Since RAM is used by MongoDB for quickly fetching data it becomes important to scale out when the active work set limits are set to reach.*
3. *And the final scenario is when the application is write intensive that Sharding can be used to spread the writes across multiple servers.*

## **Sharding Components**

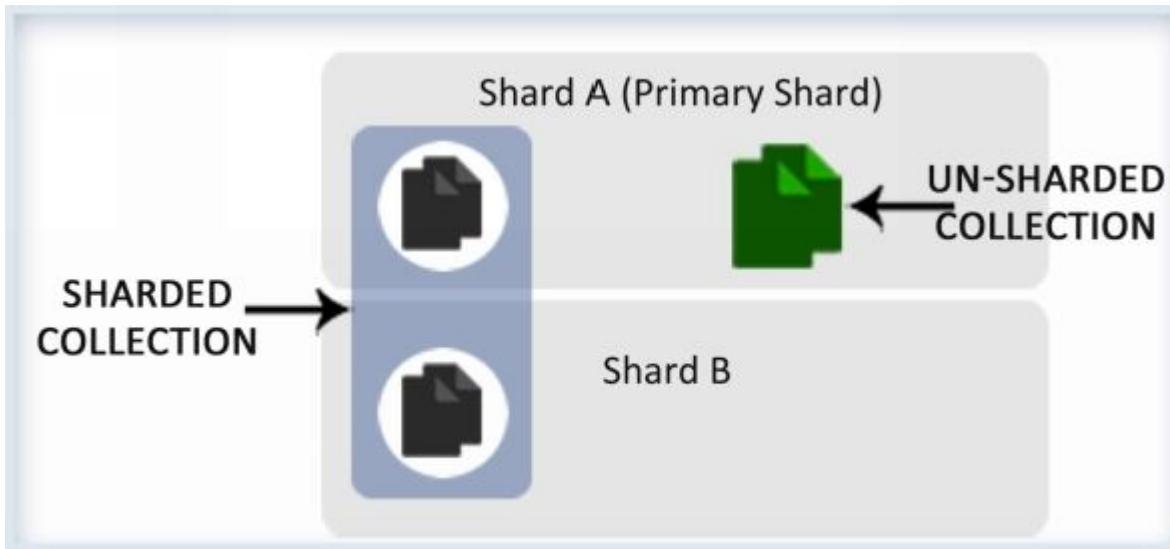
We will next look at the components that enable Sharding in MongoDB. Sharding is enabled in MongoDB via Sharded Cluster. A Sharded cluster has the following three components:

- Shards
- mongos and
- Config Servers.

**Shard** is the component where the actual data is stored. It holds a subset of data for the Sharded cluster and can either be a mongod or a replica set. All shards data combined together forms the complete dataset for the Sharded cluster.

Sharding is enabled per collection basis hence there might be collections which are not sharded. In every Sharded cluster there's a **primary shard** which holds all the unsharded collections in addition to the sharded collection data.

While deploying sharded cluster the first shard becomes the primary shard by default though it's configurable.



**Fig 7-16: Primary Shard**

**Config servers** are special mongod's that holds the metadata of the sharded cluster, where metadata reflects the state and organization of the sharded data set and system.

*Config server stores data for a single sharded cluster.* The config servers should be available for the proper functioning of the cluster.

*A single config server can lead to a single point of failure for the cluster, hence for production deployment it's recommended to have at least 3 config servers, so that the cluster keeps functioning even if one config server is not accessible.*

A Config server stores the data in the config database which enables routing of the client requests to the respective data and hence this database should not be updated.

MongoDB writes data to the config server only when the data distribution has changed for balancing the cluster.

**Mongos** – Mongos act as the routers, they are responsible for routing the read and write request from the application to the shards.

An application interacting with a mongo database need not worry about how the data is stored internally on the shards for them its transparent, it's only the mongos they interact with. The mongos in turn routes reads and writes to the shards.

Mongos cache the metadata from config server so that for every read and write request they don't overburden the config server.

*However the data is read from the config server in the following cases:*

1. *A new mongos starts for the first time, or an existing mongos restarts.*
2. *After a chunk migration, the mongos instances update themselves with the new cluster metadata. We will explain about chunk migration in detail in a while.*

## **Data distribution process**

We will next look at how the data is distributed among the shards for the collections where Sharding is enabled.

*MongoDB distributes data, or shards, at the collection level.*  
Sharding partitions a collection's data by the shard key.

## Shard Key

Shard key can be any indexed field or indexed compound field that exists in all documents of the collection. User specifies that this is the field basis which the documents of the collection need to be distributed. Internally MongoDB divides the documents based on the value of the field into chunks and distributes them across the shards.

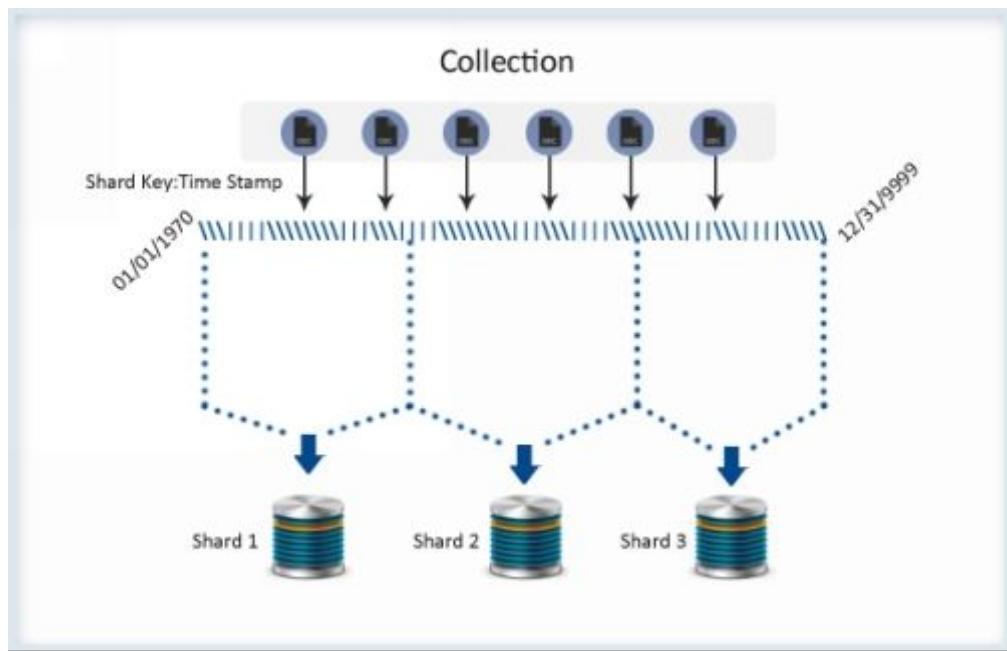
There are two ways MongoDB enables distribution of the data: first is range based partitioning and second is hash based partitioning.

### Range Based partitioning

In range based partitioning the shard key values are divided into ranges.

*Say for e.g. we consider a Timestamp field as the shard key. In this way of partitioning the values are considered as a straight line starting from a Min value to Max value where Min is the starting period say for e.g. 01/01/1970 and Max is the end period say for e.g. 12/31/9999, every document of the collection will have timestamp value within this range only and it will represent some point on the line.*

*Basis the number of shards available the line will be divided into ranges and documents will be distributed basis that.*



**Fig 7-17: Range based partitioning**

In this scheme of partitioning the documents with close shard key values are likely to be on the same shard. This can significantly improve the performance of the range queries.

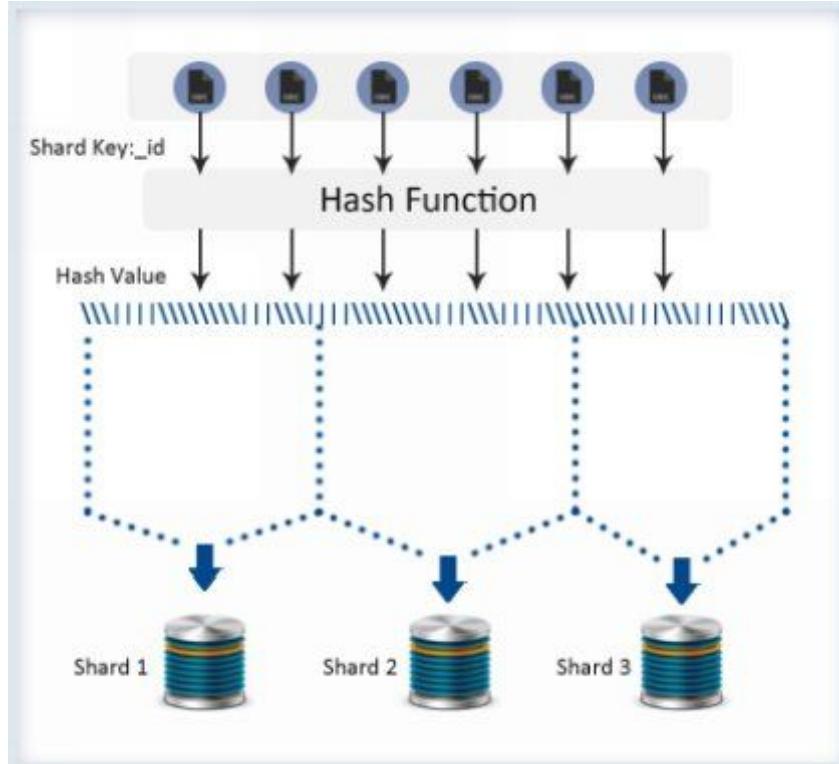
However the disadvantage is that it can lead to uneven distribution of data, overloading one of the shards which may end up receiving majority of the requests whereas the other shards remain under loaded, hence the system will not scale properly.

### Hash Based partitioning

In hash based partitioning the data is distributed on the basis of hash value of the shard field selected this will lead to a more random distribution as compared to the range based partitioning.

Hence it's unlikely that the documents with close shard key be part of the same chunk.

*For e.g. ranges based on the hash of the `_id` field. We will have a straight line of Hash values which will again be partitioned on basis of the number of shards and on basis of the hash values the documents will lie in either of the shards.*



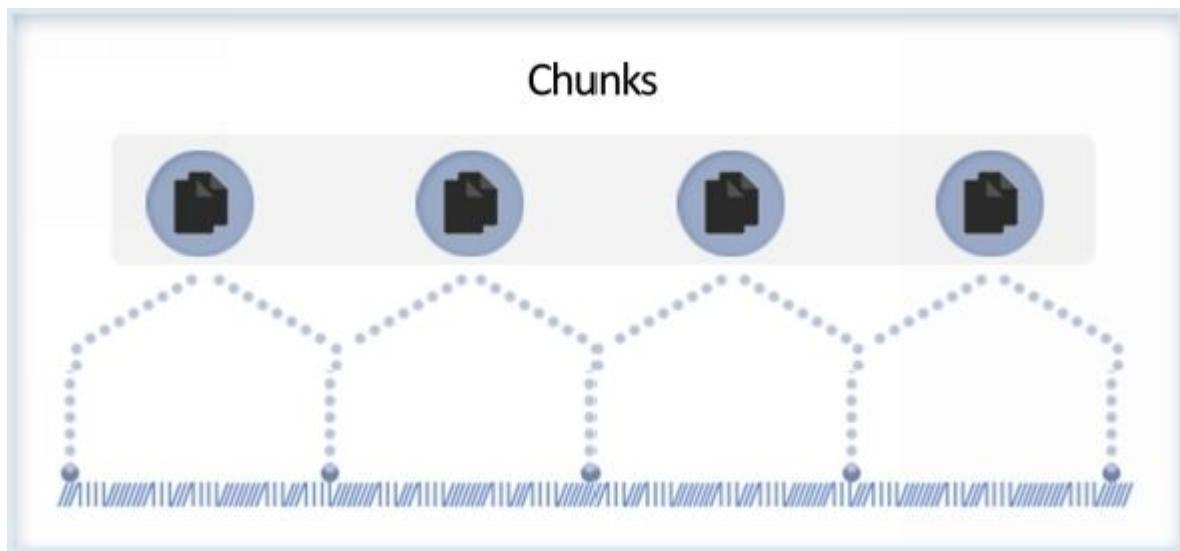
**Fig 7-18: Hash based partitioning**

Hence in contrast to the range based partitioning this ensures an even distribution of data at the expense of efficient range queries.

## Chunks

The data is moved between the shards in form of chunks. The shard key range is further partitioned into sub ranges which are also termed as chunks.

Each chunk represents a smaller range of values within the shard key's range. *The default chunk size for a sharded cluster is 64 megabytes.* In most situations, the default size is appropriate for splitting and migrating chunks.



*Fig 7-19: Chunks*

Let's discuss about the execution of sharding and chunks with an example. Say we have a collection of blog posts which is sharded on "date". This means it will be split up based on values in the "date" field. If we have three shards, they might contain data like:

- Shard #1: beginning of time up to June 2009
- Shard #2: July 2009 to November 2009
- Shard #3: December 2009 to through end of time

In order to retrieve documents from January 1<sup>st</sup> 2010 till today, the query is sent to mongos.

In this scenario this is what will happen:

1. The client queries mongos.
2. The mongos know which shards have the data so mongos sends the queries to Shard #3.
3. Shard #3 executes the query and returns the results to mongos.
4. mongos puts together the results received and sends them back to the client.

 *All of the Sharding stuff is done a layer away from the client, so the application doesn't have to be Sharding-aware, it can just query the mongos as though it were a normal mongod instance.*

Let's consider another scenario where we need to insert new document's with today's date. Here is the sequence of events:

1. You send the document to mongos.
2. Mongos checks the date and then basis on that it sends it to Shard #3.
3.  Shard #3 inserts the document.

*This is again identical to a single server setup from a client's point of view.*

## **Role of the config servers in above scenario**

Consider a scenario where we start getting millions of documents inserted with the date September 2009.

In this case the Shard #2 begins to get overloaded.

The config servers will notice this and, when shard #2 gets too big it will split the data on shard #2 and migrate some of it to other, emptier shards. Then it will let mongos know that now shard #2 contains from July 2009 to September 15th 2009 and shard #3 contains data from September 16th 2009 To end of time.

The config servers are also responsible for figuring out what to do when you add a new shard to the cluster. It figures out if it should keep the new shard in reserve or move some data to it right away.

Basically the config servers are the brains of the operation. Whenever the config servers move around data, it lets mongos know what the final configuration is so that mongos can continue routing requests correctly.

## ***Data Balancing Process***

We will next look at how the cluster is kept balanced i.e. how MongoDB ensures that all the shards are equally loaded.

The addition of new data or modification of existing data or addition or removal of servers can lead to imbalance in the data distribution which means either one shard is overloaded with more chunks whereas the other shards have less number of chunks or it can also lead to increase in the size of chunk which is significantly greater than the other chunks.

MongoDB ensures balance with two background process:

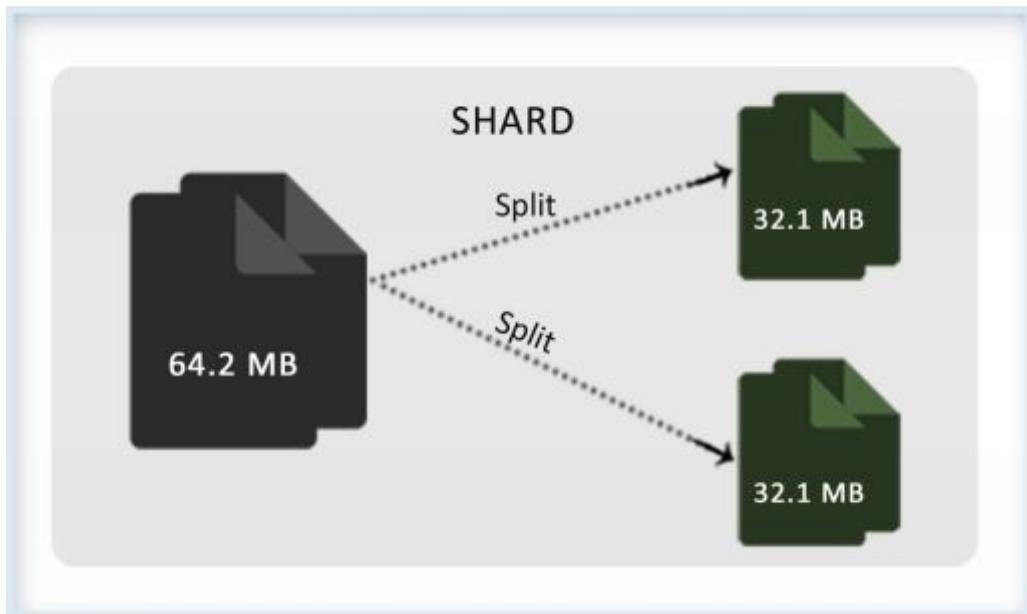
1. Chunk splitting and
2. Balancer.

## Chunk splitting

This is one of the processes which ensure the chunks are of the specified size.

As we have seen above a shard key is chosen which is used to identify how the documents will be distributed across the shards. The documents are further grouped into chunks of 64MB (default and is configurable) and are stored on the shards based on the range it is hosting.

If the size of the chunk changes due to some insert or update operation, and exceeds the default chunk size then the chunk is split into two smaller chunks by the mongos.



**Fig 7-20: Chunk splitting**

This process keeps the chunks within a shard of the specified size or lesser than that i.e. it ensures that the chunks are of the configured size.

*Insert and update operations triggers splits.* The split operation will lead to modification of the data in the config server as the metadata is modified. Though splits don't lead to migration of data however

this operation can lead to dis balance of the cluster with one shard having more chunks as compared to another.

## Balancer

Balancer is the background process which is used to ensure that all the shards are equally loaded or are in balanced state. This process manages chunks migrations.

*Splitting of the chunk is one of the reasons that can cause imbalance. In addition to this addition or removal of documents can also lead to the cluster imbalance.* In scenario of cluster imbalance, balancer is used which is the process of distributing data evenly within the cluster.

When a shard has too many chunks as compared to the other shards than MongoDB automatically balances the chunks across the shards. *This procedure is transparent to the application and the users.* Any of the mongos within the cluster can initiate the balancer process.

They do so by acquiring lock on the config database of the config server as balancer involves migration of chunks from one shard to another and hence can lead to change in the metadata and hence will lead to change in the config server database. The balancer process can have huge impact on the database performance hence it can either

- 1. Be configured to start the migration only when the migration threshold has reached. The migration threshold is the difference in number of chunks between the shards with most chunks and the shards with fewest chunks for a collection. Threshold is as follows:*

Number of Chunks	Migration Threshold
Less than 20	

**2**

**21-80**

**4**

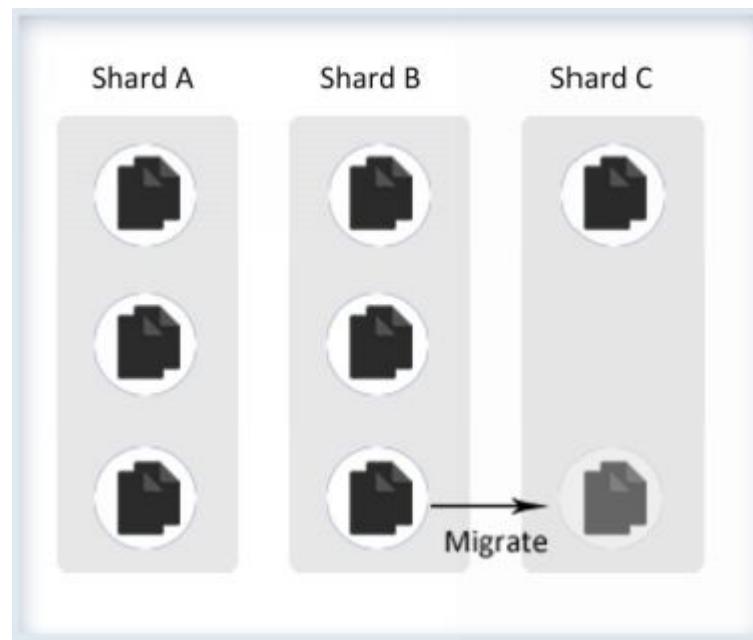
*Greater than 80*

# 8

2. *Or it can be scheduled to run in a period which will not impact the production traffic.*

The balancer migrates one chunk at a time and follows the following steps

1. moveChunk command is sent to the source shard.
2. An internal moveChunk command is started on the source where it creates the copy of the documents within the chunk and queues it. In the meanwhile any operations for that chunk are routed to the source by the mongos as the config database is not yet changed and the source will be responsible for serving any read/write request on that chunk.
3. The destination shard starts receiving the copy of the data from the source.
4. Post receiving all the documents in the chunk the destination shard starts the synchronization process to ensure that it has all changes that happened to the data during the migration process.
5. When fully synchronized, the destination shard connects to the config database and updates the cluster metadata with the new location for the chunk.
6. After the destination shard completes the update of the metadata, and once there are no open cursors on the chunk, the source shard deletes its copy of the documents.



**Fig 7-21:** *Chunk Migration*

*If the balancer needs to start additional chunk migration from the source shard it can start doing so without having to wait for the current migration process to finish this deletion step*

*If there's an error during the migration, the balancer aborts the process leaving the chunk on the origin shard. MongoDB removes the chunks data from the origin shard after the migration completes successfully.*

*Adding or removing shards can also lead to cluster imbalance. While MongoDB begins migrating data to the new shard immediately, it can take some time before the cluster balances. When removing a shard, the balancer migrates all chunks to other shards. After migrating all data and updating the meta data, the shard is removed safely.*

## **Operations**

We will next look at how the read and write operations are performed on the sharded cluster.

As mentioned above the metadata of the cluster is maintained by the Config servers which store the data in the config database. This data of the config database is used by the Mongos to service the application read and write requests.

The mongos instances cache the data and then use it for routing read and write operations to the shards, this way the config servers are not overburdened.

*The mongos will only read from the config servers in the following scenarios:*

1. *When the mongos has started for the first time or*
2. *An existing mongos has restarted or*
3. *After chunk migration when the mongos need to update its cached metadata with the new cluster metadata.*

Whenever any operation is issued the first step which the mongos need to do is to identify the shards that will be serving the request.

Since shard key is used to distribute data across the sharded cluster. Hence if the operation is using the shard key field then based on that specific shards can be targeted.

*Say for e.g. the Shard Key is employeeid*

1. *If the find query contains the employeeid field then the mongos will be able to target the query to subset of shards.*
2. *If a single update operation uses employeeid for updating the document then in that scenario also the request will be routed to the shard holding that employee data*

However if the operation is not using the shard key then the request is broadcast to all the shards. Generally a multi update or remove operations are targeted across the cluster.

*While querying the data there might be scenarios where in addition to identifying the shards and fetching the requested data from them the mongos might require to perform actions on the data before returning the final result set to the client.*

*For e.g. if the application has issued a find() request with sort(), then in this case the mongos passes the \$orderby option to the shards which returns the result set from their data set in an ordered manner. When the mongos has all the shards sorted data it performs an incremental merge sort of the results before returning the final result to the client.*

 Similar to sort we have aggregation functions, limit(), skip() etc. which require mongos to perform operations post receiving the data from the shards and before returning the final result set to the client.

Mongos has no persistent state and consumes minimal system resources and if the application requirement is simple find () queries which can be solely met by the shards and need no manipulation at the mongos level in such scenarios the mongos can be run on the same systems as your application servers.

## **Implementing Sharding**

In this section we will walk through how to setup a test configuration in a single machine on a windows platform.



*We will keep the example simple by using only two shards.*

*In this test configuration, we will use the services listed in the below table*

<b>Service</b>	<b>Daemon</b>	<b>Port</b>	<b>Dbpath</b>
<b>Shard Controller</b>	Mongos		

<b>27021</b>	N/A
<b><i>Config Server</i></b>	<i>Mongod</i>

<b>27022</b>	<i>C:\db\config\data</i>
<b><i>Shard0</i></b>	<i>Mongod</i>

<b>27023</b>	<i>C:\db\shard1\data</i>
<b><i>Shard1</i></b>	<i>Mongod</i>

**27024**

*C:\db\shard2\data*

We will be focusing on the following

1. *Setup a sharded cluster.*
2. *Create a database, collection, enable sharding on the collection.*
3. *Using import command we will load data in the sharded collection.*
4. *We will see how data is distributed amongst the shards.*
5. *We will then add, remove shards from the cluster and check how data is distributed automatically.*

## Setting the Shard Cluster

In order to set up the cluster the first step is to set up the configuration server.

Open a new terminal window and enter the following code to create the data directory for the config server and start the mongod.

```
C:\> mkdir C:\db\config\data  
C:\>CD C:\mongodb\bin  
C:\mongodb\bin>mongod --port 27022 --dbpath  
C:\db\config\data --configsvr  
Wed Feb 12 19:00:17.921 [initandlisten] MongoDB  
starting : pid=5784 port=27022  
.....  
Wed Feb 12 19:00:18.126 [initandlisten] waiting for  
connections on port 27022
```

The config server is up and running, next we need to start the mongos.

Open a new terminal window and type the following:

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongos --configdb localhost:27022 --  
port 27021 --chunkSize 1  
Wed Feb 12 19:13:33.976 warning: running with 1 config  
server should be done on!  
.....  
Wed Feb 12 19:13:46.130 [Balancer] distributed lock  
'balancer/ANOC9:27021:139221  
2613:41' unlocked.
```

We have the shard controller i.e. the mongos also up and running.

*If you look at the terminal window for the config server you will see that the shard server has connected to the config server and has registered itself with it.*

*For this example we have set the chunk size to its smallest possible size of 1 MB which is not a practical value for real-world systems because it means that the chunk storage is smaller than the maximum size of a document (4 MB).*

*However, this is just a demonstration, and the small chunk size allows you to create a lot of chunks to exercise the sharding setup without having to load a lot of data. By default, chunkSize is set to 128 MB unless otherwise specified.*

Next we need to bring up the two shard servers.

Open fresh terminal window. Create the data directory for the first shard and start the mongod

```
C:\>mkdir C:\db\shard0\data  
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongod --port 27023 --dbpath  
c:\db\shard0\data --shardsvr  
Wed Feb 12 19:23:34.107 [initandlisten] MongoDB  
starting : pid=6200 port=27023  
Wed Feb 12 19:23:34.372 [initandlisten] waiting for  
connections on port 27023
```

Open fresh terminal window. Create the data directory for the second shard and start the mongod

```
C:\>mkdir c:\db\shard1\data  
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongod --port 27024 --dbpath  
C:\db\shard1\data --shardsvr  
Wed Feb 12 19:25:12.552 [initandlisten] MongoDB  
starting : pid=5692 port=27024  
Wed Feb 12 19:25:12.788 [websvr] admin web console  
waiting for connections on port 28024
```

We have all the servers up and running. Next we need to tell the sharding system where the shard servers are located. In order to do this we need to connect to the shard controller i.e. mongos.

 It's important to remember that, even though mongos is not a full MongoDB instance, it appears to be a full instance to your application. Hence we can use the mongo shell to attach to the shard controller i.e. mongos and perform any operation on the mongos.

Open the mongos mongo console

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongo localhost:27021  
MongoDB shell version: 2.4.9  
connecting to: localhost:27021/test  
mongos>
```

Switch to admin database

```
mongos> use admin  
switched to db admin  
mongos>
```

Add the shards information by running the following commands

```
mongos>  
db.runCommand({addshard:"localhost:27023",allowL  
ocal:true})  
{ "shardAdded" : "shard0000", "ok" : 1 }  
mongos>  
db.runCommand({addshard:"localhost:27024",allowL  
ocal:true})  
{ "shardAdded" : "shard0001", "ok" : 1 }  
mongos>
```

This activates the two shard servers.

*The next command can be used to check the shards*

```
mongos> db.runCommand({listshards:1})
```

```
{  
  "shards" : [  
    {  
      "_id" : "shard0000",  
      "host" : "localhost:27023"  
    }, {  
      "_id" : "shard0001",  
      "host" : "localhost:27024"
```

```
        }
    ], "ok" : 1
}
```

## Create database and shard collection

*In order to continue further with the example we will be creating a database testdb and then create a collection testcollection which we will be sharding on the key testkey.*

Connect to the mongos console and issue the following command to get the database.

```
mongos> testdb=db.getSiblingDB("testdb")
testdb
```

Next we will be enabling sharding at database level for testdb

```
mongos> db.runCommand({enablesharding: "testdb"})
{ "ok" : 1 }
mongos>
```

In the next step we will be specifying the collection which needs to be sharded and the key on which the collection will be sharded.

```
mongos>         db.runCommand({shardcollection:
"testdb.testcollection", key: {testkey:1}})
{ "collectionsharded" : "testdb.testcollection", "ok" : 1 }
mongos>
```

With the completion of the above steps we now have a sharded cluster setup with all components up and running, we have also created a database and enabled sharding on the collection.

Next we will be importing data into the collection so that we can check how the data is distributed on the shards.

We will be using the import command to load data in the testcollection.

Open a new terminal window and issue the following command.

```
C:\>cd C:\mongodb\bin
C:\mongodb\bin>mongoimport --host ANOC9 --port
27021 --db testdb --collection testcollection --type
```

```
csv --file c:\mongoimport.csv --headerline  
connected to: ANOC9:27021  
Thu Feb 13 11:56:52.001 Progress: 1040288/1777806  
58%  
Thu Feb 13 11:56:52.001 62500 20833/second  
Thu Feb 13 11:56:53.562 check 9 100001  
Thu Feb 13 11:56:53.582 imported 100000 objects  
The mongoimport.csv consists of two fields. First is the testkey which  
is a random number and the second field is a text field. The purpose  
of the second field is to ensure these documents occupy a sufficient  
number of chunks and hence make it feasible to use the Sharding  
mechanism.
```

This will insert 100,000 records with random testkeys and some **testtext** to the documents.

*In order to vet whether the records are inserted or not, connect to the mongos console and issue the following command*

```
C:\Windows\system32>cd c:\mongodb\bin  
c:\mongodb\bin>mongo localhost:27021  
MongoDB shell version: 2.4.9  
connecting to: localhost:27021/test  
mongos> use testdb  
switched to db testdb  
mongos> db.testcollection.count()
```

# 100000

*mongos>*

We will next connect to the two shards i.e. shard0 and shard1 console and look at how the data is distributed. Open a new terminal window and connect to shard0's console.

C:\>cd C:\mongodb\bin

C:\mongodb\bin>mongo localhost:27023

*MongoDB shell version: 2.4.9*

*connecting to: localhost:27023/test*

Switch to testdb and issue the count() command to check number of documents on the shard.

> use testdb

*switched to db testdb*

> db.testcollection.count()

## **42518**

Next open a new terminal window and connect to shard1's console, and follow the steps as above i.e. switch to testdb and check the count of testcollection collection.

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongo localhost:27024  
MongoDB shell version: 2.4.9  
connecting to: localhost:27024/test  
> use testdb  
switched to db testdb  
> db.testcollection.count()
```

## 57482

>

 We may see different values for the number of documents in each shard depending on time we are checking the shards i.e. the number of documents in a given shard may vary from moment to moment reason being the mongos instance may initially place all the chunks on one shard, but over time it will rebalance the shard set to evenly distribute data among all the shards by moving chunks around.

### Adding a New Shard to the Cluster

We have a sharded cluster set up and we have also sharded a collection and looked at how the data is distributed amongst the shards. Next we will add a new shard server to the cluster to spread out the load a little more.

Addition of a shard is easy we will be repeating the steps as mentioned above. Begin by opening a new terminal window and creating a data directory for the new shard.

```
c:\>mkdir c:\db\shard2\data
```

Next we will start the mongod at port 27025.

```
c:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongod --port 27025 --dbpath  
C:\db\shard2\data --shardsvr
```

```
Thu Feb 13 12:29:45.847 [initandlisten] MongoDB  
starting : pid=7060 port=27025
```

.....

```
Thu Feb 13 12:29:46.113 [websvr] admin web console  
waiting for connections on port 28025
```

We will now add the new shard server to the cluster. In order to configure we will open a terminal window and connect to the mongos console.

```
C:\>cd c:\mongodb\bin
```

```
c:\mongodb\bin>mongo localhost:27021
MongoDB shell version: 2.4.9
connecting to: localhost:27021/test
mongos>
```

Switch to admin database and issue the addshard command to add the shard server to the sharded cluster.

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard:
  "localhost:27025", allowlocal: true})
{ "shardAdded" : "shard0002", "ok" : 1 }
mongos>
```

*Run the listshards command to vet whether the shard has been added to the cluster or not.*

```
mongos> db.runCommand({listshards:1})
{
  "shards" : [
    {
      "_id" : "shard0000",
      "host" : "localhost:27023"
    },
    {
      "_id" : "shard0001",
      "host" : "localhost:27024"
    },
    {
      "_id" : "shard0002",
      "host" : "localhost:27025"
    }
  ],
  "ok" : 1
}
```

Next we will check how the testcollection data is distributed. Open a new terminal window and connect to the new shard's console.

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongo localhost:27025  
MongoDB shell version: 2.4.9  
connecting to: localhost:27025/test
```

Switch to testdb and check the collections listed on the shard.

```
> use testdb  
switched to db testdb  
> show collections  
system.indexes  
testcollection
```

Issue count of the testcollection subsequently three times

```
> db.testcollection.count()  
6928  
> db.testcollection.count()
```

**12928**

> db.testcollection.count()

## 16928

*Interestingly the number of items in the collection is slowly going up. Reason being the mongos is rebalancing the data across the expanded cluster.*

*Over time, the sharding system will migrate chunks from the shard0 and shard1 storage servers to create an even distribution of data across the three servers that make up the cluster.*

*This process is automatic, and it will happen even if there is no new data being inserted into the testcollection collection. In this case, the mongos shard controller is moving chunks to the new server, and then registering them with the config server.*

*This is one of the factors to consider when choosing a chunk size.*

*If chunkSize value is very large, then we will get a less even distribution of data across the shards; conversely, smaller the chunkSize value, the more even the distribution of data will be.*

## Removing a Shard from the Cluster

In the next example, we will remove the shard server which we added in the previous example.

In order to initiate the process we need to log on to the mongos console, switch to admin db and issue the following command to remove the shard from the shard cluster.

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongo localhost:27021  
MongoDB shell version: 2.4.9  
connecting to: localhost:27021/test  
mongos> use admin  
switched to db admin  
mongos> db.runCommand({removeShard:  
"localhost:27025"})  
{  
  "msg" : "draining started successfully",  
  "state" : "started",  
  "shard" : "shard0002",  
  "ok" : 1  
}  
mongos>
```

*The removeShard command responds with a message indicating that the removal process has started.*

*It also indicates that the shard controller (mongos) has begun relocating the chunks on the target shard server to the other shard servers in the cluster. This process is known as draining the shard.*

The progress of the draining process can be checked by reissuing the `removeShard` command.

```
mongos> db.runCommand({removeShard:  
"localhost:27025"})  
{  
  "msg" : "draining ongoing",  
  "state" : "ongoing",  
  "remaining" : {  
    "chunks" : NumberLong(2),  
    "dbs" : NumberLong(0)  
  },  
  "ok" : 1  
}  
mongos>
```

*The response will tell you how many chunks and databases still need to be drained from the shard. If we reissue the command and the process is terminated, the output of the command will depict the same.*

```
mongos> db.runCommand({removeShard:  
"localhost:27025"})  
{  
  "msg" : "removeshard completed successfully",  
  "state" : "completed",  
  "shard" : "shard0002",  
  "ok" : 1  
}  
mongos>
```

*In order to verify whether the `removeShard` command was successful, the `listshards` command can be used.*

 Next we can terminate the Shard2 mongod process and delete the storage files as the data is already successfully migrated to the

other shards.

*The ability to add and remove shards to the shard cluster without having to take it offline is a critical component of MongoDB's ability to support highly scalable, highly available, large-capacity data stores.*

## **Listing the Status of a Sharded Cluster**

`printShardingStatus()` command is used for dumping the status of the sharded cluster. This command gives lots of insights into the internals of the sharding system.

```
mongos> db.printShardingStatus()
--- Sharding Status ---
sharding version: {
    "_id" : 1,
    "version" : 3,
    "minCompatibleVersion" : 3,
    "currentVersion" : 4,
    "clusterId" : ObjectId("52fb7a8647e47c5884749a1a")
}
shards:
{ "_id" : "shard0000", "host" : "localhost:27023" }
{ "_id" : "shard0001", "host" : "localhost:27024" }
.....
```

The output lists the shard servers, the configuration of each sharded database/collection, and each chunk in the sharded dataset.

Since we have used a small chunkSize value to simulate a large sharding setup that's why the report lists lot of chunks.

 *An important piece of information that can be obtained from this listing is the range of sharding keys associated with each chunk.*

*The preceding output also shows which shard server the specific chunks are stored on. We can use the output returned by this command as the basis for a tool to analyze the distribution of a shard*

*server's keys and chunks. For example, we might use this data to determine whether there is any clumping of data in the dataset.*

## **Controlling collection distribution (Tag Based Sharding)**

In the previous section we have seen how the data is distributed across various shards based on the shard key. *Shard tagging is a new feature in MongoDB version 2.2.0.*

Tagging gives operators control over which collections go to which shard. It is used to force writes to go to a local data center, but it can also be used to pin a collection to a shard or set of shards.

In order to understand tag based sharding let us set up a sharded cluster. We will be using the shard cluster created above. For this example we need 3 shards, hence we will add shard2 again to the cluster.

### **Prerequisite**

We will start the cluster first. Just to reiterate we need to follow the following steps for doing the same.

**Step 1:** Start the config server. Open a terminal window and enter the following command (if it's not already running)

```
C:\> mkdir C:\db\config\data  
C:\>CD C:\mongodb\bin  
C:\mongodb\bin>mongod --port 27022 --dbpath  
C:\db\config\data --configsvr
```

**Step 2:** Start the mongos. Open a new terminal window and enter the following (if it's not already running)

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongos --configdb localhost:27022 --  
port 2702
```

**Step 3:** We will start the shard servers next

Start Shard0. Open a terminal window and enter the following command (if it's not already running)

```
C:\>cd c:\mongodb\bin
```

```
c:\mongodb\bin>mongod --port 27023 --dbpath  
c:\db\shard0\data --shardsvr
```

Start Shard1. Open a terminal window and enter the following command (if it's not already running)

```
C:\>cd c:\mongodb\bin  
C:\mongodb\bin>mongod --port 27024 --dbpath  
c:\db\shard1\data --shardsvr
```

Start Shard2. Open a terminal window and enter the following command (if it's not already running)

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongod --port 27025 --dbpath  
c:\db\shard2\data --shardsvr
```

Since we have removed the Shard2 from the sharded cluster in the earlier example above, hence it's essential that before proceeding further we need to add Shard2 to the sharded cluster because for this example we will need three shards.

In order to do so we need to connect to the mongos. Open a terminal window and enter the following command

```
C:\Windows\system32>cd c:\mongodb\bin  
c:\mongodb\bin>mongo localhost:27021  
MongoDB shell version: 2.4.9  
connecting to: localhost:27021/test  
mongos>
```

*Before we add the shard to the cluster we need to delete the testdb database.*

```
mongos> use testdb  
switched to db testdb  
mongos> db.dropDatabase()  
{ "dropped": "testdb", "ok": 1 }  
mongos>
```

Next add the shard “Shard2” to the cluster using the following steps

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard:
"localhost:27025", allowlocal: true})
{ "shardAdded" : "shard0002", "ok" : 1 }
mongos>
```

 *Note: If you try adding the removed shard without removing the testdb database, it will give error.*

```
mongos> db.runCommand({addshard:
"localhost:27025", allowlocal: true})
{
  "ok" : 0,
  "errmsg" : "can't add shard localhost:27025 because a
local database 'testdb' exists in another
shard0000:localhost:27023"}
```

In order to ensure that all the three shards are present in the cluster, run the following command.

```
mongos> db.runCommand({listshards:1})
{
  "shards" : [
    {
      "_id" : "shard0000",
      "host" : "localhost:27023"
    },
    {
      "_id" : "shard0001",
      "host" : "localhost:27024"
    },
    {
      "_id" : "shard0002",
      "host" : "localhost:27025"
    }
  ]
}
```

```
}
```

```
],
```

```
"ok" : 1}
```

## Tagging

By the end of the above steps we have our sharded cluster with 3 shards, a config server and a mongos up and running. Next we will start up a new shell and connect to the mongos at port 30999 and config db at 27022.

```
C:\ >cd c:\mongodb\bin
```

```
c:\mongodb\bin>mongos --port 30999 --configdb
```

```
localhost:27022
```

*Thu Feb 13 16:08:33.009 warning: running with 1 config server should be done only for testing purposes and is not recommended for production*

*.....*

*Thu Feb 13 16:08:33.297 [Balancer] distributed lock 'balancer/ANOC9:30999:1392287913:41' unlocked.*

Next start up a new shell, connect to the mongos and enable sharding on the collections.

```
C:\ >cd c:\mongodb\bin
```

```
c:\mongodb\bin>mongo localhost:27021
```

*MongoDB shell version: 2.4.9*

*connecting to: localhost:27021/test*

```
mongos> show dbs
```

*admin (empty)*

```
config 0.234375GB
```

```
testdb 0.203125GB
```

```
mongos> conn=new Mongo("localhost:30999")
```

*connection to localhost:30999*

```
mongos> db=conn.getDB("movies")
```

```
movies
```

```
mongos> sh.enableSharding("movies")
```

```

{ "ok" : 1 }
mongos>          sh.shardCollection("movies.drama",
{originality:1})
{ "collectionsharded" : "movies.hindi", "ok" : 1 }
mongos>          sh.shardCollection("movies.action",
{distribution:1})
{ "collectionsharded" : "movies.action", "ok" : 1 }
mongos>          sh.shardCollection("movies.comedy",
{collections:1})
{ "collectionsharded" : "movies.comedy", "ok" : 1 }
mongos>

```

The steps followed are:

1. Connect to the mongos console
2. View the running databases connected to the mongos instance running at port 30999
3. Get reference to the database “movies”
4. Enable sharding of the database “movies”
5. Sharded the collection movies.drama by shard key “originality”
6. Sharded the collection movies.action by shard key “distribution”
7. Sharded the collection movies.comedy by shard key “collections”

Next let's insert some data in the collections, using the following sequence of commands

```

mongos>for(var i=0;i<100000;i++)
{db.drama.insert({originality:Math.random(), count:i,
time:new Date()});}
mongos>for(var i=0;i<100000;i++)
{db.action.insert({distribution:Math.random(),
count:i, time:new Date()});}

```

```
mongos>for(var i=0;i<100000;i++)
{db.comedy.insert({collections:Math.random(),
count:i, time:new Date()});}
mongos>
```

By the end of the above step we have three shards and three collections with sharding enabled on the collections. Next we will see how data is distributed across the shards.

Switch to config db

```
mongos> use config
switched to db config
mongos>
```

We can use chunks.find to look at how the chunks are distributed.

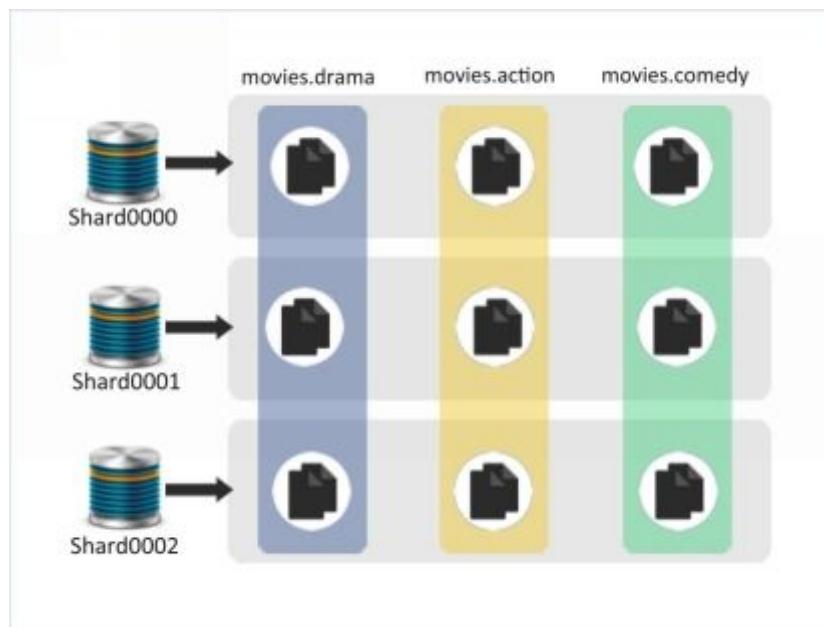
```
mongos> db.chunks.find({ns:"movies.drama"},  
{shard:1, _id:0}).sort({shard:1})  
{ "shard" : "shard0000" }  
{ "shard" : "shard0001" }  
{ "shard" : "shard0001" }  
{ "shard" : "shard0001" }  
{ "shard" : "shard0002" }  
{ "shard" : "shard0002" }  
{ "shard" : "shard0002" }  
  
mongos> db.chunks.find({ns:"movies.action"},  
{shard:1, _id:0}).sort({shard:1})  
{ "shard" : "shard0000" }  
{ "shard" : "shard0001" }  
{ "shard" : "shard0002" }  
{ "shard" : "shard0002" }  
{ "shard" : "shard0002" }
```

```

mongos>      db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos>

```

*As we can see from the result the chunks are pretty evenly spread out amongst the shards.*



**Fig 7-22: Distribution without tagging**

Next we will use tags to separate the collections. Intent of this is to have one collection per shard i.e. our goal is to have the chunk distribution as follows:

Collection Chunks	on shard
<b>movies.drama</b>	Shard0000
<b>movies.action</b>	Shard0001
<b>movies.comedy</b>	Shard0002

 A **tag** describes a property of a shard, the property can be any property hence you might tag a shard as “fast” or “slow” or “east coast” or “rack space”.

In this example we will tag the shards as belonging to each of the collection

```
mongos> sh.addShardTag("shard0000", "dramas")
mongos> sh.addShardTag("shard0001", "actions")
mongos> sh.addShardTag("shard0002", "comedies")
mongos>
```

This means the following:

Put any chunks tagged “dramas” on shard0000. Put any chunks tagged “actions” on shard0001. And put any chunks tagged “comedies” on shard0002.

Next we will create rules to tag the collections chunk accordingly.

**Rule 1:** All chunks created in the movies.drama collection will be tagged as “dramas”.

```
mongos> sh.addTagRange("movies.drama",
{originality:MinKey}, {originality:MaxKey}, "dramas")
mongos>
```

*This means, “Mark every chunk in movies.drama with the ‘dramas’ tag” (MinKey is negative infinity, MaxKey is positive infinity, so all of the chunks fall in this range).*

Similar to this we will make rules for the other two collections also.

**Rule 2:** All chunks created in movies.action collection will be tagged as “actions”

```
mongos> sh.addTagRange("movies.action",
{distribution:MinKey}, {distribution:MaxKey},
"actions")
mongos>
```

**Rule 3:** All chunks created in movies.comedy collection will be tagged as “comedies”

```
mongos>          sh.addTagRange("movies.comedy",  
{collection:MinKey},           {collection:MaxKey},  
"comedies")
```

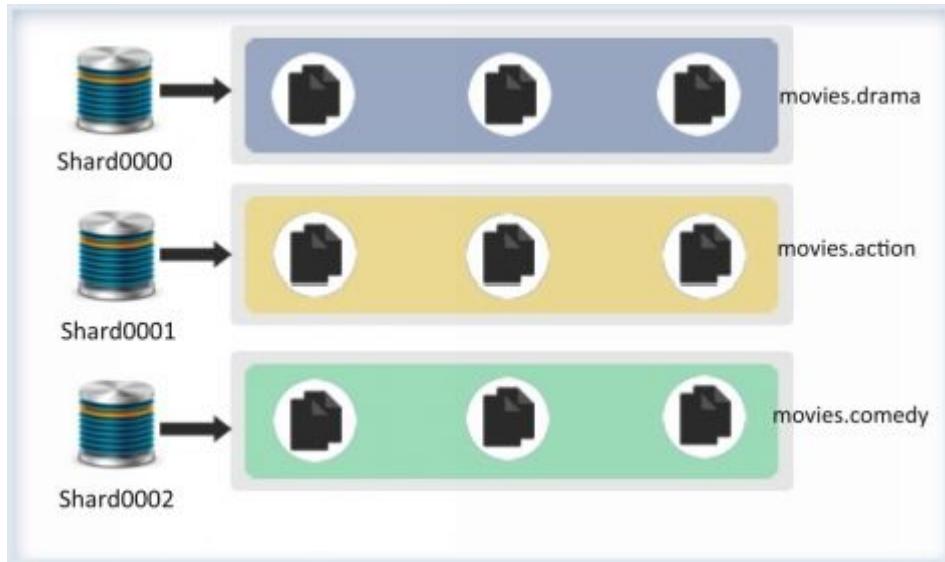
*mongos>*

We need to wait for the cluster to rebalance so that the chunks are distributed based on the tags and rules defined above. As we have mentioned above the chunk distribution is an automatic process, hence after some time the chunks will automatically be redistributed to implement the changes we have done.

Next issue chunks.find to vet the chunks organization.

```
{ "shard" : "shard0000" }
```





**Fig 7-23: Distribution with tagging**

Thus the collection chunks have been redistributed based on the tags and rules defined.

## Scale with Tagging

Next we will look at how we can scale with tagging. Let's change the scenario, let's assume the collection `mongos.action` needs two servers for its data. Since we have only three shards this means the other two collections data need to be moved to one shard.

Hence in this scenario we will change the tagging of the shards. We will add the tag "comedies" to shard0 and remove the tag from shard2, and further add the tag "actions" to shard2.

This implies that the chunks tagged "comedies" will be moved to shard0 and chunks tagged "actions" will be spread to shard2.

We will first move the collection `movies.comedy` chunk to shard0 and remove the same from shard2.

```
mongos> sh.addShardTag("shard0000","comedies")
mongos>
sh.removeShardTag("shard0002","comedies")
```

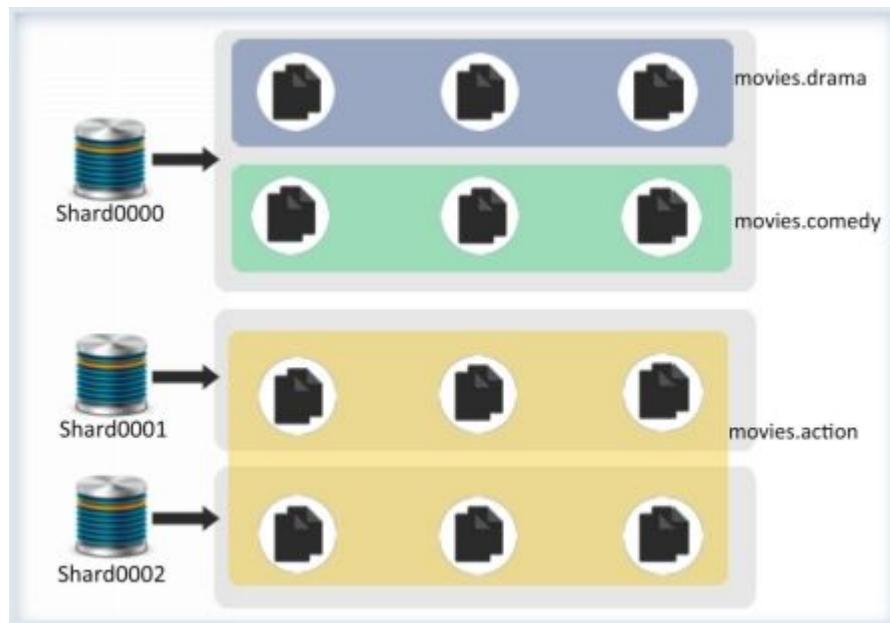
Next we will add the tag "actions" to shard2, so that `movies.action` chunks spread across shard2 also.

```
mongos> sh.addShardTag("shard0002","actions")
```

Re-issuing the find command after some time will show the following results

```
mongos> db.chunks.find({ns:"movies.drama"},  
{shard:1, _id:0}).sort({shard:1})  
{ "shard" : "shard0000" }  
mongos> db.chunks.find({ns:"movies.action"},  
{shard:1, _id:0}).sort({shard:1})  
{ "shard" : "shard0001" }  
{ "shard" : "shard0002" }
```

```
mongos>      db.chunks.find({ns:"movies.comedy"},  
{shard:1, _id:0}).sort({shard:1})  
{ "shard" : "shard0000" }  
mongos>
```



**Fig 7-24: Tagging with scaling**

*The chunks have been redistributed reflecting the changes made.*

## Multiple Tags

We can have multiple tags associated with the shards. Let's add two different tags to the shards.

*Say for e.g. we want to distribute the writes based on the disk say we have one shard which has spinning disk and the other is having SSD hence we want to redirect 50% of the writes to the SSD shard and remaining to the spinning disk.*

The same will be achieved as below. We will first tag the shards basis this property.

```
mongos> sh.addShardTag("shard0001", "spinning")
mongos> sh.addShardTag("shard0002", "ssd")
mongos>
```

Let's further assume we have "distribution" field of the movies.action collection which we will be using as the shard key. The value of the "distribution" field is between 0 and 1, so we want to say, "If distribution < .5, send this to the spinning disk. If distribution >= .5, send to the SSD." Hence we will define the rules as follows

```
mongos>sh.addTagRange("movies.action",
{distribution:MinKey} ,{distribution:.5} , "spinning")
mongos>sh.addTagRange("movies.action" ,
{distribution:.5} ,{distribution:MaxKey}, "ssd")
mongos>
```

Hence the documents with distribution < .5 will be written to spinning shard whereas the others will be written to the SSD disk shard.

*Hence with tagging as we add new servers, we can control what kind of load they get.*

## ***Points to remember when importing data in a sharded environment***

### **Pre-split your data**

This requires some knowledge about the data you're going to import. So instead of leaving MongoDB the choice of which chunks to create you have to do it yourself. Anyway you will end up telling MongoDB where to split your chunks like so:

```
db.runCommand( { split : "mydb.mycollection" ,  
    middle : { shardkey : value } } );
```

After you have done this you can also tell MongoDB which of those chunks should reside on which node. Again this depends on your use case and how data is read by your application. So keep things like data locality in mind.

### **Increase chunk size**

The documentation has the following to say on chunk sizes:

1. Small chunks lead to a more even distribution of data at the expense of more frequent migrations, which creates expense at the query routing (mongos) layer.
2. Large chunks lead to fewer migrations, which is more efficient both from the networking perspective and in terms of internal overhead at the query routing layer. Large chunks produce these efficiencies at the expense of a potentially more uneven distribution of data.

Pick a good shard key - A very essential point for a good distribution among nodes.

## ***Sharding and Monitoring***

In most cases the components of sharded clusters benefit from the same monitoring and analysis as all other MongoDB instances. Additionally, clusters require monitoring to ensure that data is effectively distributed among nodes and that Sharding operations are functioning appropriately.

## **Config Servers**

The config database provides a map of documents to shards. The cluster updates this map as chunks move between shards.

When a configuration server becomes inaccessible, some Sharding operations like moving chunks and starting mongos instances become unavailable.

However, clusters remain accessible from already-running mongos instances.

Because inaccessible configuration servers can have a serious impact on the availability of a sharded cluster, we should monitor the configuration servers to ensure that the cluster remains well balanced and that mongos instances can restart.

## **Balancing and Chunk Distribution**

The most effective sharded cluster deployments require that chunks are evenly balanced among the shards.

MongoDB has a background balancer process that distributes data such that chunks are always optimally distributed among the shards.

Issue the `db.printShardingStatus()` or `sh.status()` command to the mongos by way of the mongoshell.

This returns an overview of the entire cluster including the database name, and a list of the chunks.

## **Stale Locks**

In nearly every case, all locks used by the balancer are automatically released when they become stale. However, because any long lasting lock can block future balancing, it's important to insure that all locks are legitimate. To check the lock status of the database, connect to a mongos instance using the mongo shell.

Issue the following command sequence to switch to the config database and display all outstanding locks on the shard database:

```
use config  
db.locks.find()
```

For active deployments, the above query might return a useful result set.

The balancing process, which originates on a randomly selected mongos, takes a special “balancer” lock that prevents other balancing activity from transpiring.

Use the following command, also to the config database, to check the status of the “balancer” lock.

```
db.locks.find( { _id : "balancer" } )
```

If this lock exists, make sure that the balancer process is actively using this lock.

## Production Cluster Architecture

We have discussed about Sharding and replica set. In this section we will look at how the production cluster architecture should be.

*In order to understand this lets consider a very generic use case of Friend Social networking application where the users can create there circle of friends and can share their comments or pictures across the group. The user can also comment or like their friend's comments or pictures. The users are geographically distributed.*

Hence the application requirement is immediate availability across geographies of all the comments, data should be redundant so that the user's comments, posts and pictures are not lost and it should be highly available.

Hence the application's production cluster should have the following components:

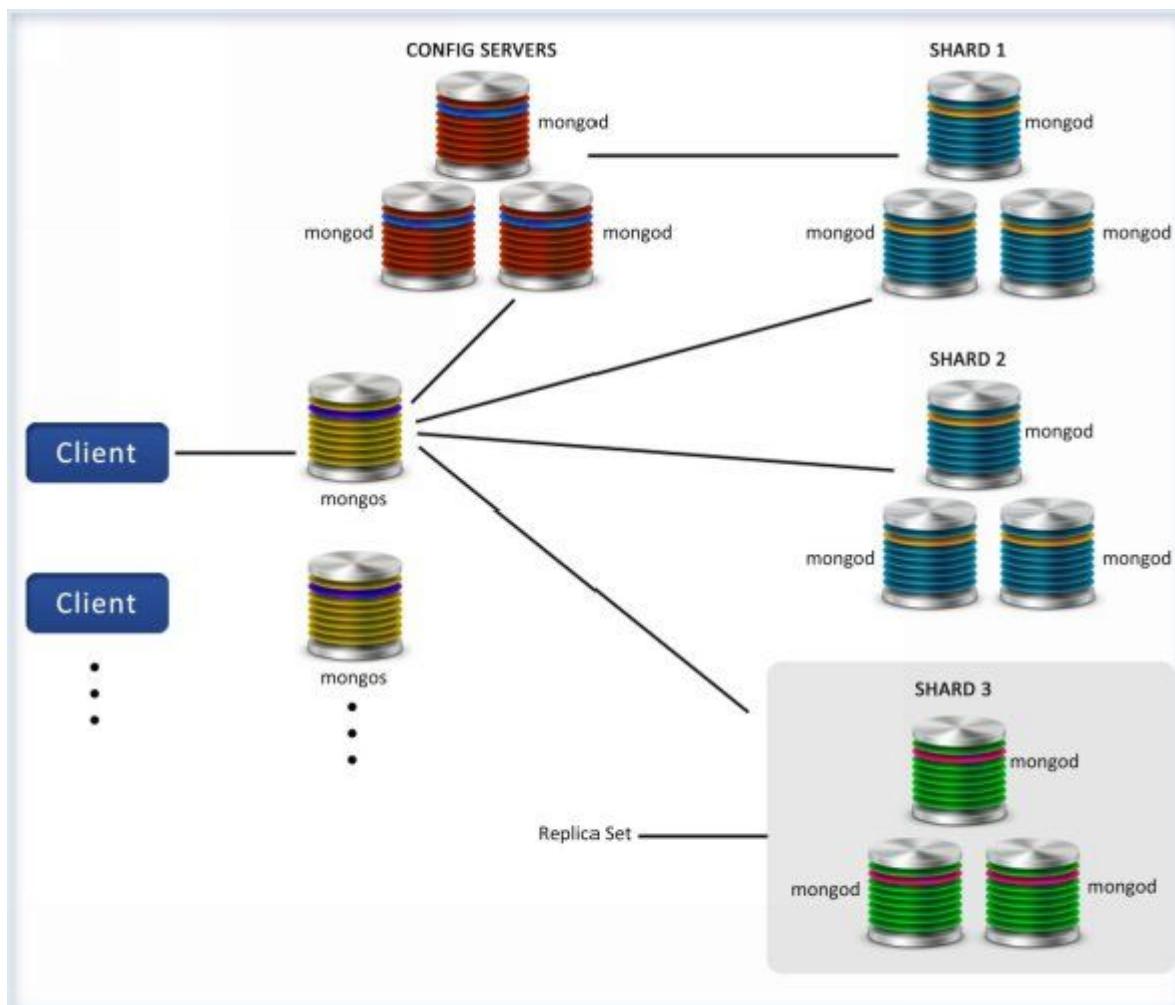
1. Two or more [mongos](#) instances. Typically, a single [mongos](#) instance is deployed on each application server. Alternatively,

you may deploy several [mongos](#) nodes and let your application connect to these via a load balancer.

2. Three [config servers](#), each residing on a discrete system.

A single [sharded cluster](#) must have exclusive use of its [config servers](#). If you have multiple shards, you will need to have a group of config servers for each cluster.

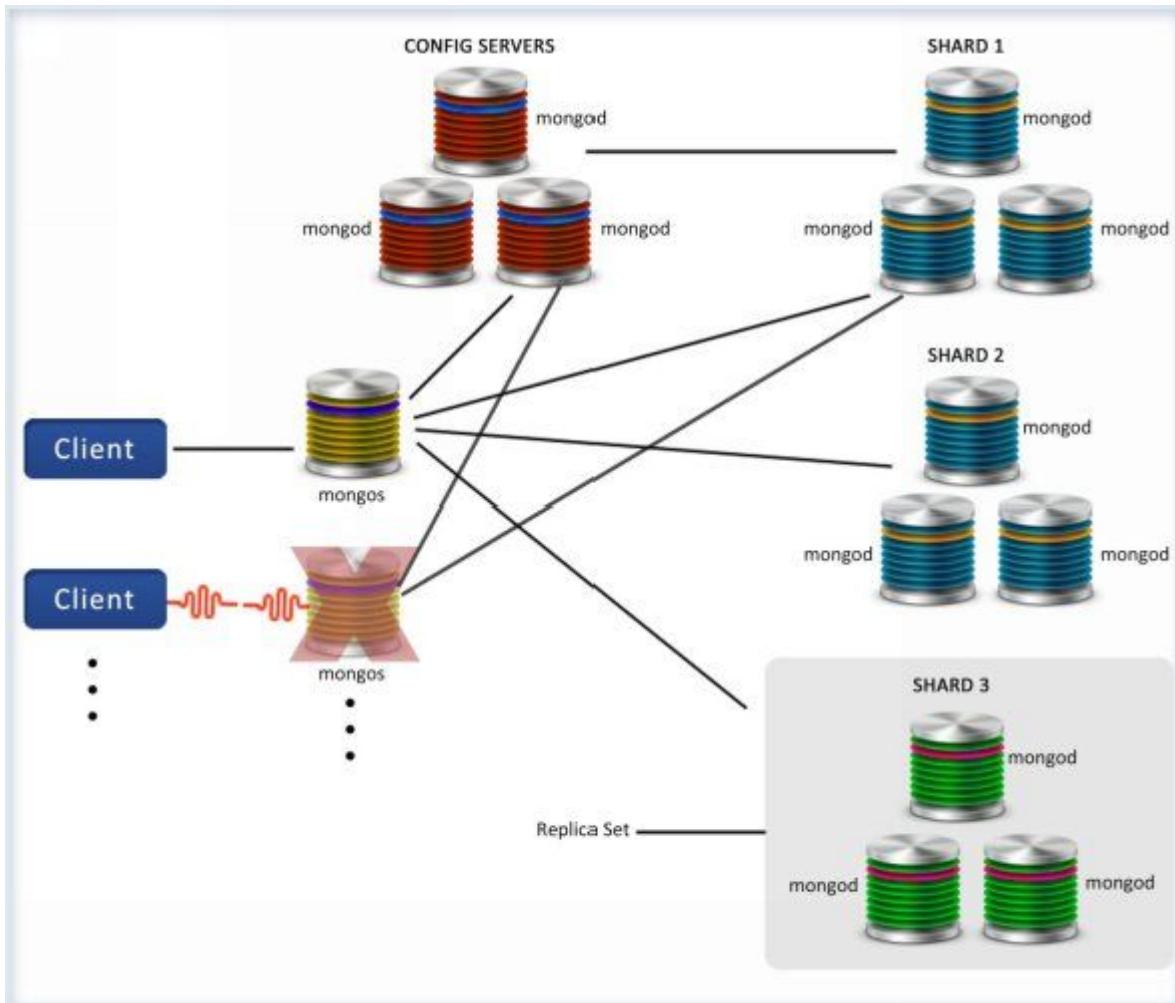
3. Two or more [replica sets](#) to serve as [shards](#). The replica sets are distributed across geographies with read concern set to nearest



**Fig 7-25: Production cluster architecture**

Before we conclude the section we will look at what could be possible failure scenarios in MongoDB production deployment and its impact on the environment

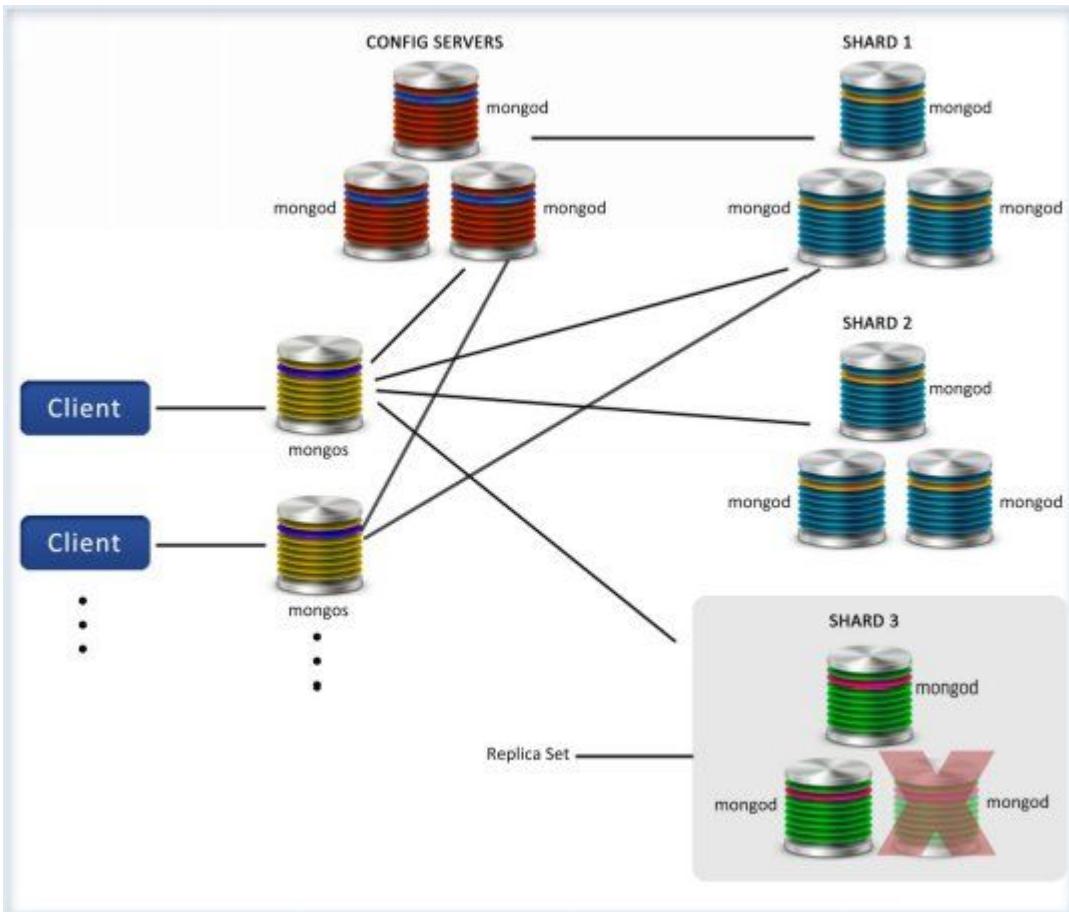
## Scenario 1



*Fig 7-26: mongos become unavailable*

**Mongos become unavailable** – The application server where mongos has gone down will not be able to communicate with the cluster however it will not lead to any data loss as the mongos don't maintain any data of its own. The mongos can restart and while restarting it can sync up with the config servers to cache the cluster metadata and the application can normally starts its operations.

## Scenario 2

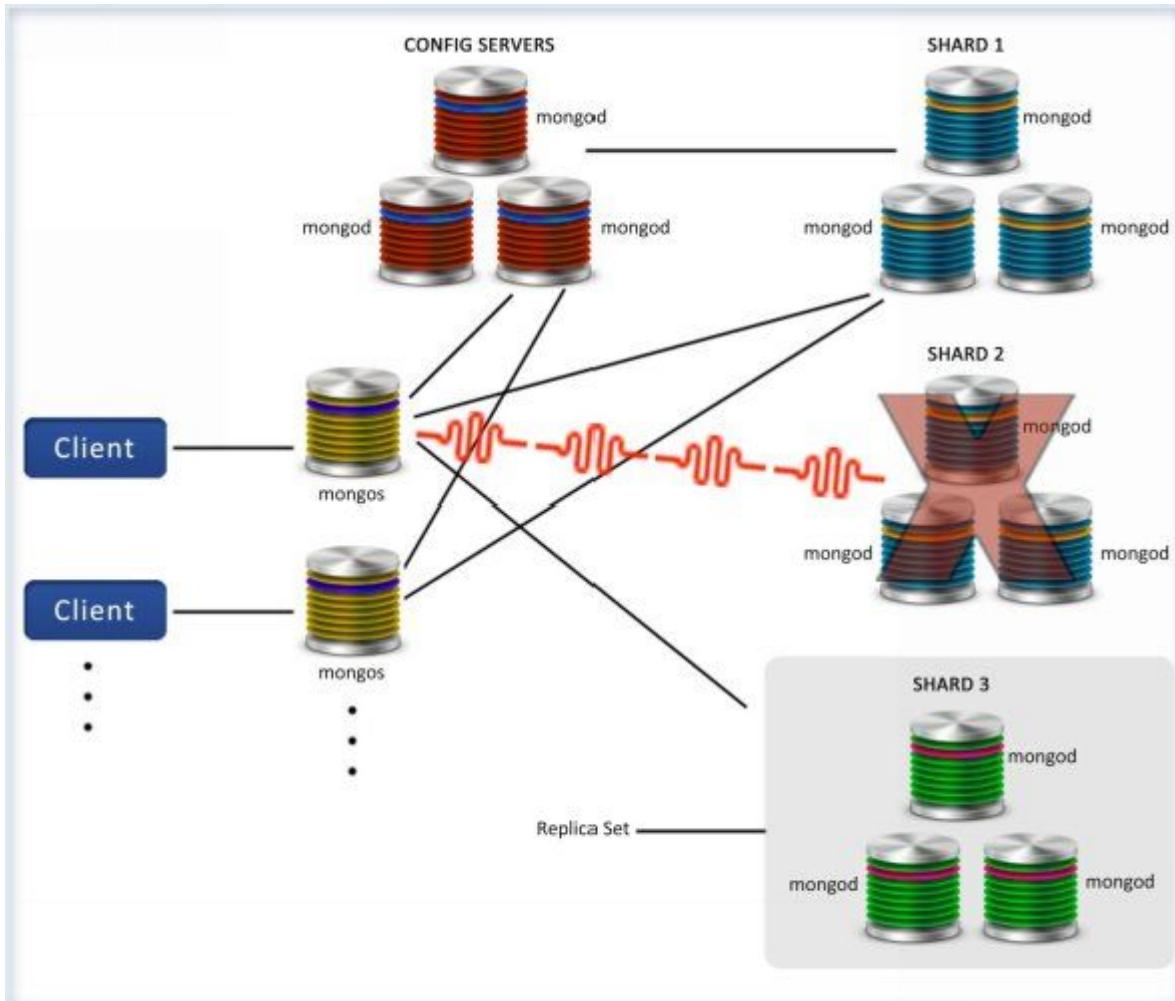


*Fig 7-27: one of the mongod of replica set is unavailable*

**One of the mongod of the replica set becomes unavailable in a shard –** Since we have used replica sets to provide high availability hence it ensures the data is not lost. If the mongod which is down is a primary node then the replica set chooses a new primary whereas if it's a secondary node then it is disconnected and the functioning continues normally.

The only difference will be the duplication of the data will be reduced making the system little weak hence we should in parallel check if the mongod is recoverable and if it is then it should be recovered and restarted whereas if it's unrecoverable, replace it and create a new member of the replica set as soon as possible to replace the lost redundancy.

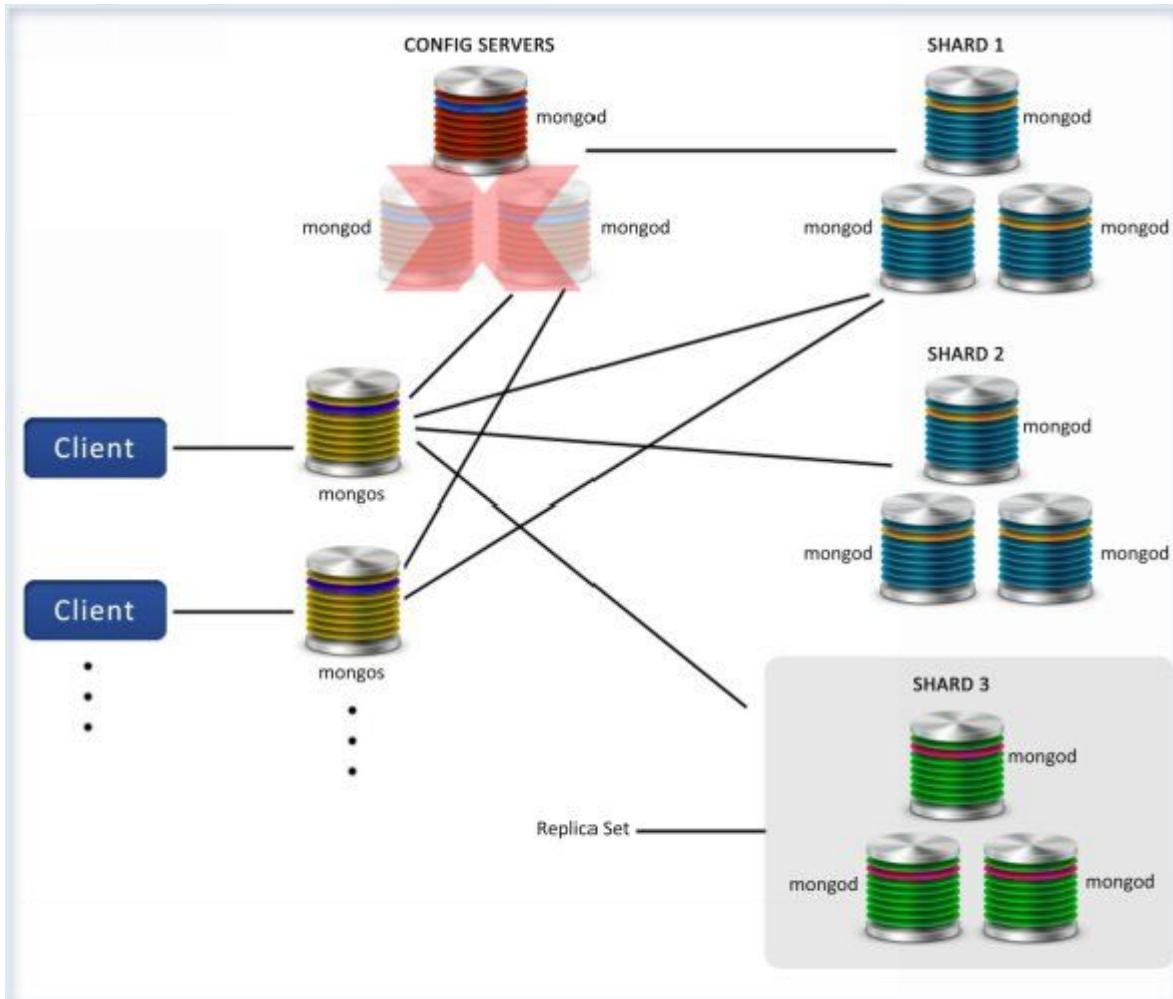
## Scenario 3



**Fig 7-28: shard unavailable**

**If one of the shard becomes unavailable** - In this scenario the data which is there on the shard will be unavailable however the other shards will be available hence it'll not stop the application, the application can still continue to read and write from other shards, however the application must deal with partial results. In parallel the shard should be attempted to recover as soon as possible.

## Scenario 4



*Fig 7-29: only one config server available*

**Only one config server is available out of three** – In this scenario though the cluster will become read only. It will not serve any operations that might lead to change in the cluster structure hence leading to change of metadata such as chunk migration or chunk splitting.

The config servers should be replaced as soon as possible. If all the config servers are unavailable the cluster will become inoperable.

## Data Storage Model

In the previous section we looked at the core services that are deployed as part of MongoDB and also looked at replica sets and

sharding. In this section we will briefly understand what's happening under the hood.

MongoDB uses mmap storage engine as its default storage engine which uses Memory Mapped files.

Memory mapped files are the files with data that the Operating system places in memory by way of the mmap() system call. mmap is an OS feature that maps a file on disk into virtual memory. We need to remember that the virtual memory is not equivalent to physical memory. A virtual memory is space on the computer's hard disk that is used in conjunction with physical RAM.

MongoDB uses memory mapped files for managing and interacting with data. MongoDB memory maps data files to memory as it accesses documents.

Data that isn't accessed is not mapped to memory. It allows the OS to control the memory mapping and allocate the maximum amount of RAM.

## Data File

First we will understand what a Data File is. As we have seen under the core services the default data directory which is used by a mongod is /data/db/.

Under this directory for every database there are separate files. Each database has single .ns file and several data files, which have monotonically increasing numeric extensions.

*For e.g. if we create a database mydbpoc, it will be stored in the files mydbpoc.ns, mydbpoc.1, mydbpoc.2 and so on.*

Name	Date modified	Type	Size
admin	2/14/2014 4:09 PM	File Folder	
journal	2/18/2014 5:21 PM	File Folder	
local	12/9/2013 7:39 PM	File Folder	
mydb	12/9/2013 7:39 PM	File Folder	
mydbnew	2/14/2014 4:09 PM	File Folder	
mydbpoc	2/14/2014 4:10 PM	File Folder	
products	2/14/2014 4:10 PM	File Folder	
admin.0	2/14/2014 4:09 PM	0 File	65,536 KB
admin.1	2/14/2014 4:09 PM	1 File	131,072 KB
admin.ns	2/14/2014 4:09 PM	NS File	16,384 KB
local.0	2/14/2014 4:09 PM	0 File	65,536 KB
local.ns	2/14/2014 4:09 PM	NS File	16,384 KB
mongod.lock	2/18/2014 5:21 PM	LOCK File	0 KB
mydb.0	2/14/2014 4:09 PM	0 File	65,536 KB
mydb.1	2/14/2014 4:09 PM	1 File	131,072 KB
mydb.ns	2/14/2014 4:09 PM	NS File	16,384 KB
mydbnew.0	2/14/2014 4:09 PM	0 File	65,536 KB
mydbnew.1	2/14/2014 4:09 PM	1 File	131,072 KB
mydbnew.ns	2/14/2014 4:09 PM	NS File	16,384 KB
mydbnnm.0	2/17/2014 6:33 PM	0 File	65,536 KB
mydbnnm.1	2/17/2014 6:33 PM	1 File	131,072 KB
mydbnnm.ns	2/17/2014 6:33 PM	NS File	16,384 KB
mydbpoc.0	2/14/2014 4:09 PM	0 File	65,536 KB
mydbpoc.1	2/14/2014 4:09 PM	1 File	131,072 KB
mydbpoc.2	2/14/2014 4:09 PM	2 File	262,144 KB
mydbpoc.ns	2/14/2014 4:09 PM	NS File	16,384 KB
products.0	2/14/2014 4:10 PM	0 File	65,536 KB
products.1	2/14/2014 4:10 PM	1 File	131,072 KB
products.ns	2/14/2014 4:10 PM	NS File	16,384 KB

**Fig 7-30: Data files**

The numeric data files for a database will double in size for each new file, up to a maximum file size of 2GB. This behavior is by design and *this behavior allows small databases to not waste too much space on disk, while keeping large databases in mostly contiguous regions on disk.*

We also need to note that MongoDB preallocates data files to ensure consistent performance.

This behavior can be disabled using the --noprealloc option. Preallocation happens in the background and is initiated every time that a data file is filled.

*This means that the MongoDB server will always attempt to keep an extra, empty data file for each database to avoid blocking on file allocation.*

Next we will see how the data is actually stored under the hood. Doubly linked list is the key data structure which is used for storing the data.

## Namespace (.ns file)

Within the data files we have data space divided into namespaces where the namespace can correspond to either a collection or an index.

The metadata of these namespaces are stored in the .ns file (If you will check your data directory you will find a file [dbname].ns).

The size of the .ns file which is used for storing the metadata is 16 MB. This file can be assumed as a big hash table which is partitioned into small buckets which are of approximately 1 KB size.

Each bucket stores metadata specific to a namespace.



**Fig 7-31:** Namespace data structure

## Collection Namespace

A collection namespace bucket contains metadata such as

1. Name of the collection
2. Few statistics of the collection such as count, size etc.  
(This is the reason whenever a count is issued against the collection it returns quick response.)
3. Index details - maintains links to each index created.
4. A deleted list
5. A doubly linked list storing the extent details i.e. it stores pointer to the first and the last extent.



**Fig 7-32: Collection Namespace details**

## Extent

Extent refers to a group of data records within a data file hence a group of extents forms the complete data for a namespace.

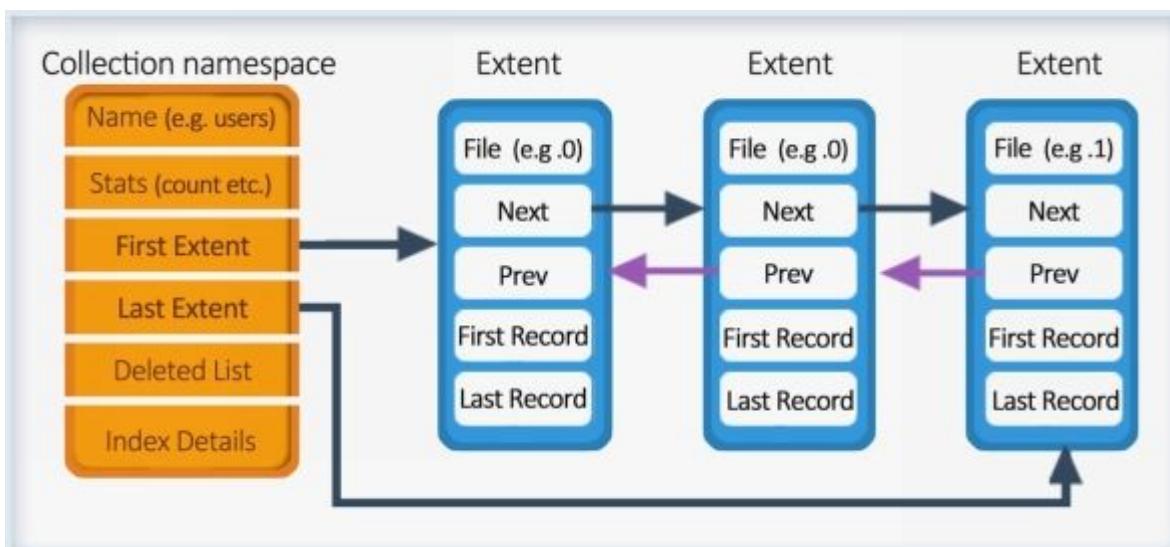
Extent uses disk locations to refer to the location on the disk where the data is actually residing. It consists of two parts - file number and offset.

File number specifies the data file it's pointing to for e.g. 0, 1, etc.

Offset is the position within the file i.e. how deep within the file we need to look for the data. Offset size is of 4B. Hence the offset's maximum value can be up to  $2^{31}-1$  which if you will see above is the maximum file size the data files can grow to i.e. 2048MB (2 GB).

An Extent data structure consists of the following things

1. Location on the disk which is the file number its pointing to
2. Since an extent is stored as a doubly linked list element hence it has a pointer to the next and the previous extent.
3. Once it has the file number it's referring to, the group of the data records within the file which it's pointing to are further stored as doubly linked list. Hence it maintains pointer to the first data record and the last data record of the data block it's pointing to which are nothing but the offsets within the file i.e. how deep within the file the data is stored.



**Fig 7-33: Extent**

## Data Record

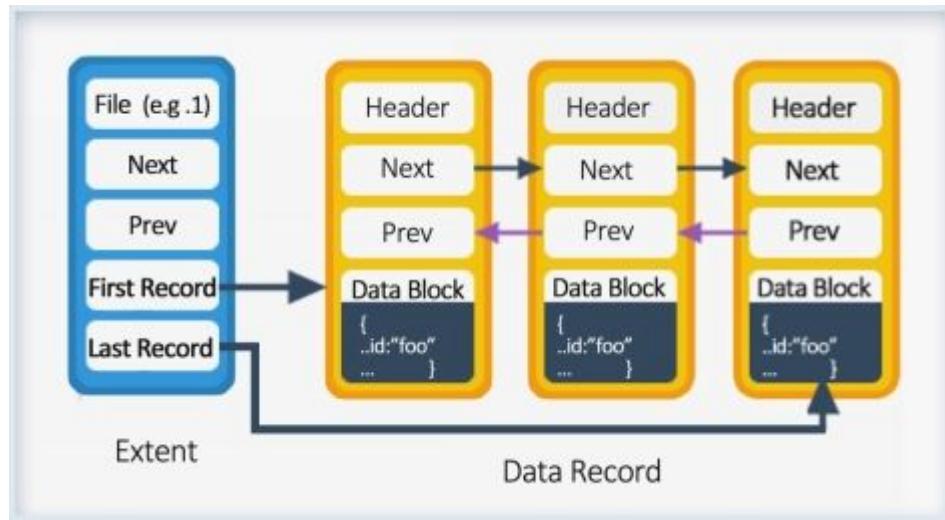
Next we will look at the data record structure. The data structure consist of the following details

1. Since the data record structure is element of the extents doubly linked list hence it stores information of the previous and the next record.
2. It has length with headers
3. And finally the data block.

The data block can have either a BTree Bucket (in case of index namespace) or a BSON object. We will be looking into the BTree structure in a while.

The BSON Object corresponds to the actual data for the collection. We need to note that the size of BSON object need not be same as the data block.

This design decision is useful to avoid movement of an object from one block to another whenever an update leads to change in the object size. Padding factor is used by MongoDB for allocation of space say for e.g. if we have a document of 3K and the padding factor specified is 1.5K then the space which will be allocated for the file will be 4.5K. If padding factor is 1 that means the allocated space as well as the object space matches. This is basically the case when we have an insert only data where the size is fixed and never varies. The padding factor is local to each namespace and the maximum it can go up to 2.



**Fig 7-34: Record Data structure**

## Deleted List

The deleted list stores details of the extent whose data has been deleted or moved (movement whenever an update has caused change in size leading to non-fitment of data in the allocated space).

The size of the record determines the bucket in which the free extent need to be placed, basically this are bucketed single linked list. When a new extent is required for fitment of data for the namespace it will first search the free list to check whether any appropriate size extent is available.

## In Summary

Hence we can assume the data files (files with numeric extensions) to be divided across different collection namespaces where extents of the namespace specifies the range of data from the data file belonging to that respective collection.

Having understood how the data is stored we will next see how a db.users.find() works:

It will first check the mydbpoc.ns file to reach to the users namespace, find out the first extent it's pointing to, it'll follow the first extent link to the first record following the next record pointer it will read the data records of the first extent till the last record is reached then it follows the next extent pointer to read it's data records in the

similar fashion and this is followed till the last extent data records are read.

## \$freelist

The .ns file has a special namespace called \$freelist for extents. \$freelist keep track of the extents which are no longer used for e.g. extents of a dropped collection or index.

## Indexes BTree

We have looked at how the collection data is stored we will next look at how the indexes are stored. The BTree structure is used for storing the indexes.

A BTree looks as below,

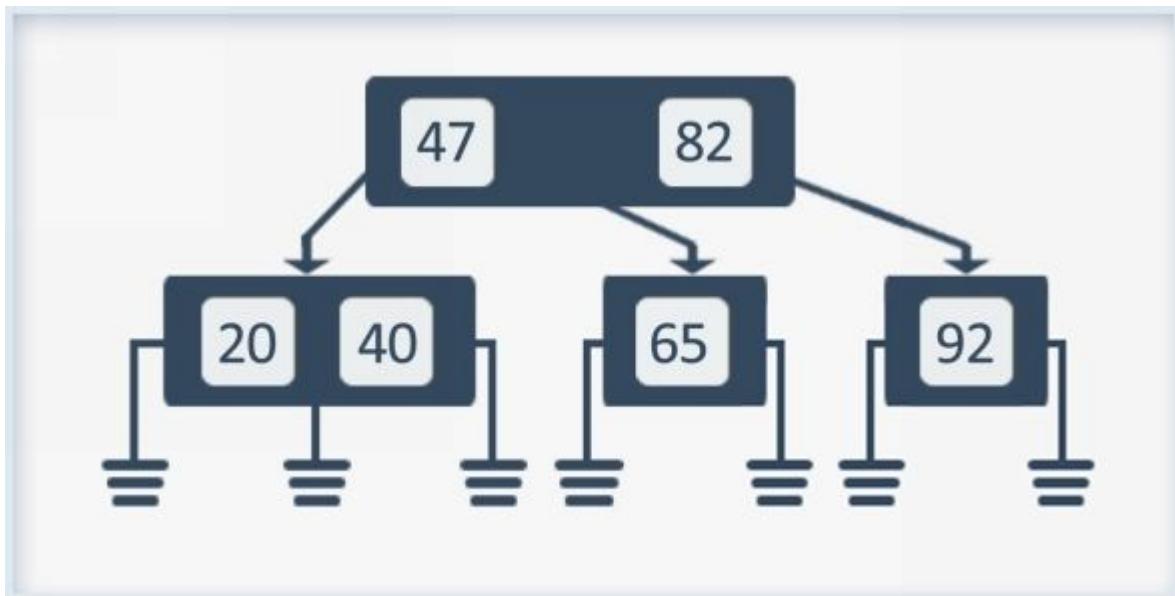
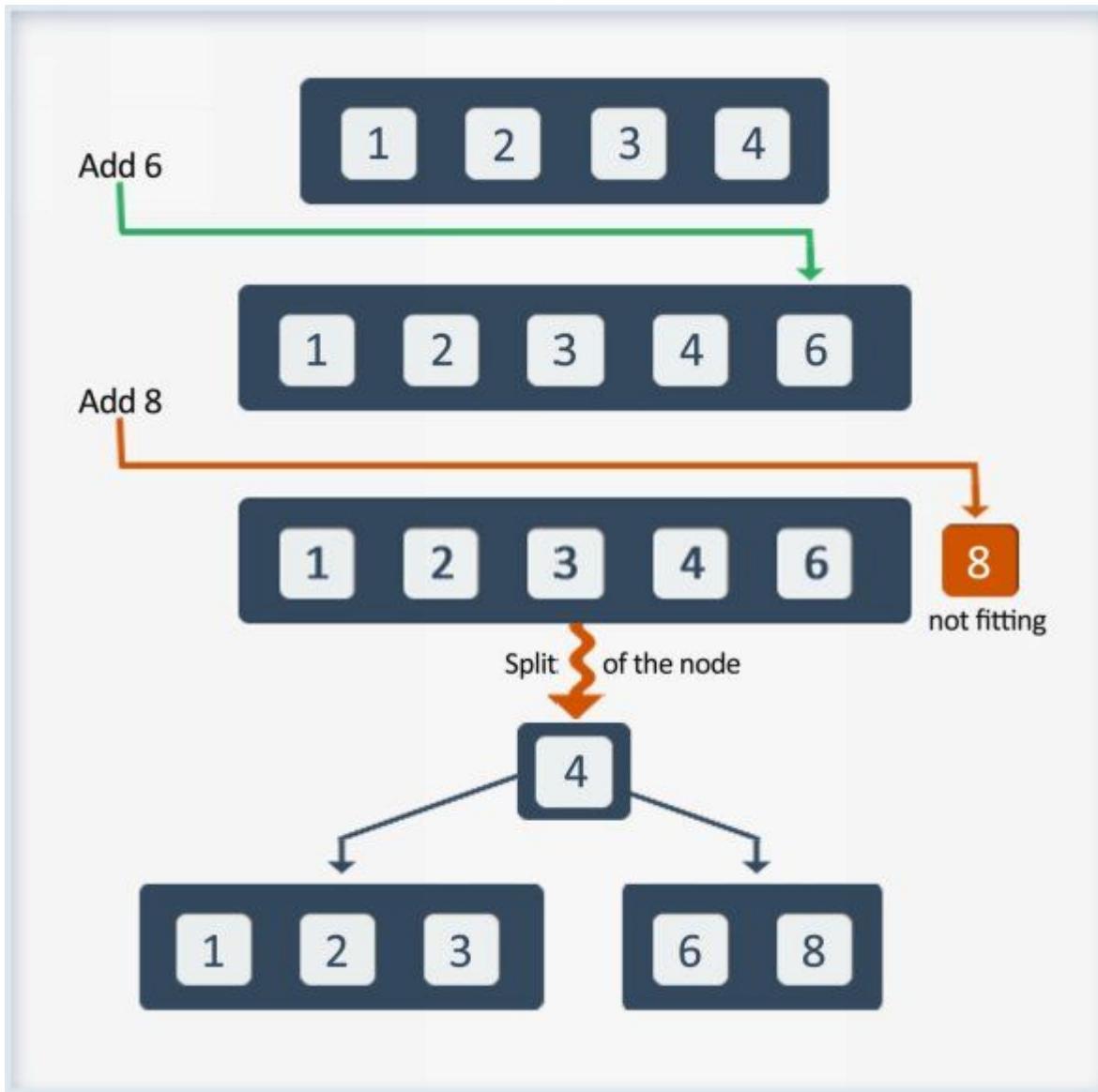


Fig 7-35: BTree

In a standard implementation of BTree whenever a new key is inserted in a BTree the default behavior is as follows.

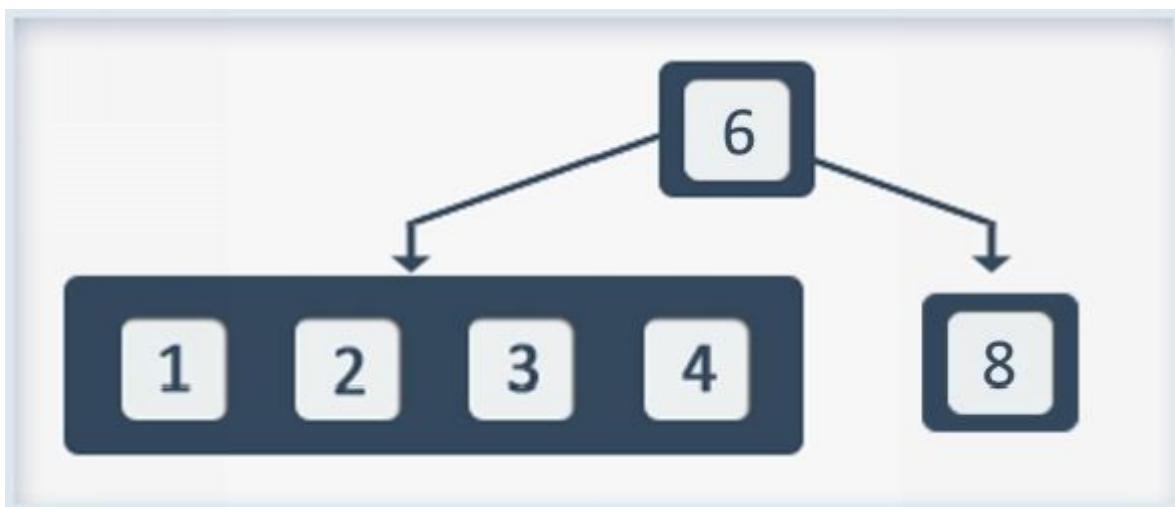


*Fig 7-36: B-Tree standard implementation*

There's a slight variation in the way MongoDB implements the BTree.

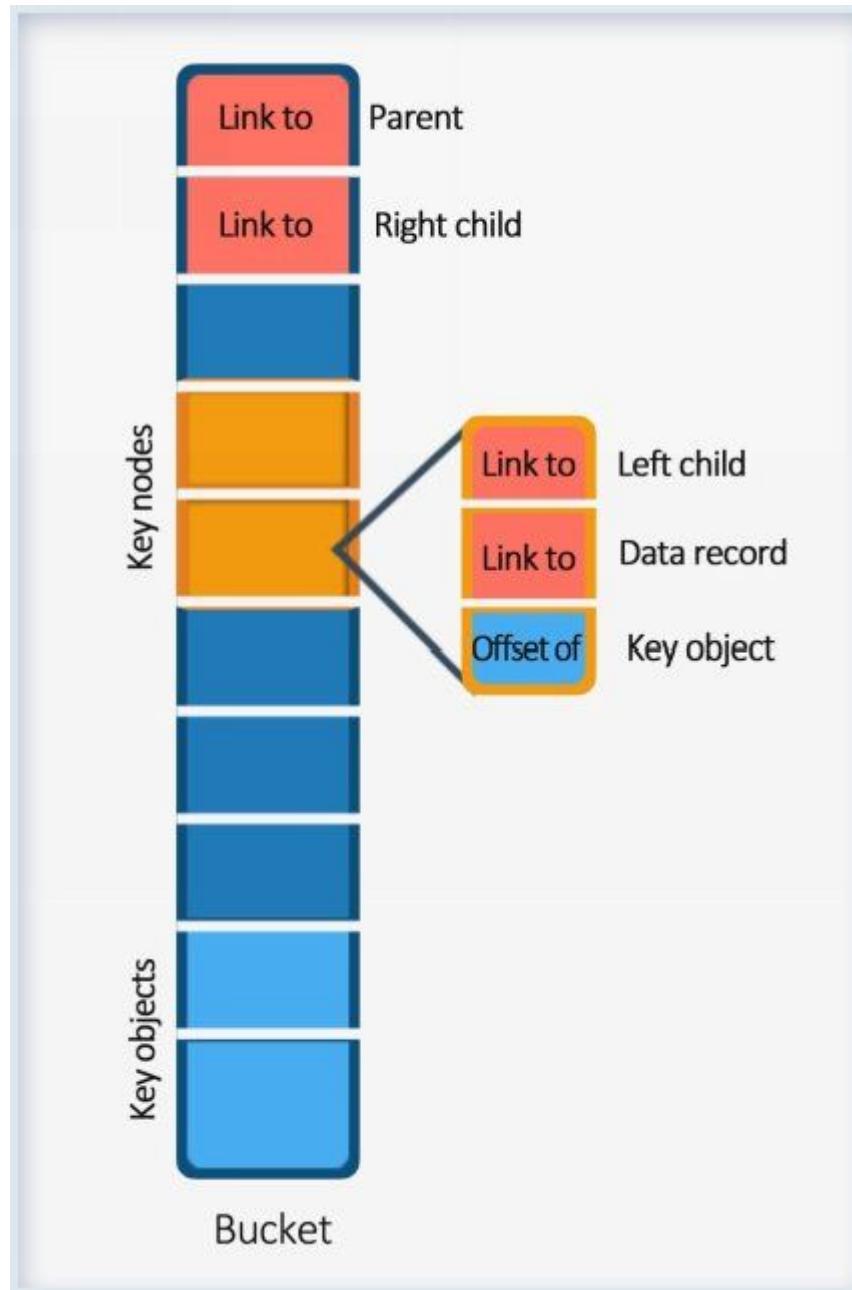
In the above scenario if we have keys such as Timestamp, ObjectId or an incrementing number then the buckets will always be half filled hence leading to lots of space wastage.

Hence in order to overcome this MongoDB has modified this slightly - whenever it identifies that the index key is an incrementing key then rather than doing a 50/50 split it does a 90/10 split as shown below



**Fig 7-37:** MongoDB's B-Tree 90/10 split

We will next see the bucket structure. Each bucket of the BTree is of 8KB.



**Fig 7-38: BTree bucket data structure**

The bucket consists of

1. Pointer to the parent
2. Pointer to the right child

3. Pointer to key nodes.
4. And List of key objects – These objects are of varying size and are stored in an unsorted manner. These objects are actually the value of the index keys.

## Key Nodes

Key nodes are nodes of fixed size and are stored in a sorted manner. It enables easy split and movement of the elements between different nodes of the BTree.

A key node contains

1. pointer to the left child,
2. the data record the index key belongs to
3. And a key offset. - The key offset is the offset of the key objects which basically tell where in the bucket the key's value is stored.

# Read and writes

We will briefly look at how the reads and writes are happening. As we have mentioned above when MongoDB updates and reads from the DB it is actually reading and writing to RAM.

In case the modification increases the size of record beyond its originally allocated space, the whole record will be moved to a bigger region with some extra padding bytes.

The padding bytes are used as a growth buffer so that future expansion doesn't require moving the data again.

The amount of padding is dynamically adjusted per collection based on its modification statistics.

On the other hand, the space occupied by the original doc will be freed up. This is tracked by a list of free lists of different sizes as mentioned above it's the \$freelist namespace in the .ns file.

 As we can imagine fragmentation will occur over time as objects are created, deleted or modified, this fragmentation will hurt performance as less data is being read/write per disk I/O. Therefore, we need to run the "compact" command periodically, which copies the data to a contiguous space.

This "compact" operation however is an exclusive operation and has to be done offline. Typically this is done in a replica set by rotating each member to offline mode one at a time to perform the compaction.

The files in RAM are flushed to disk every 60 seconds. To prevent data loss in the event of power failure, the default is to run with journaling switched on.

The journal file is flushed to disk every 100ms and if there is power loss is used to bring the database back to a consistent state. We will be discussing in a while how write happens with journaling.

All indexes on the documents in the database are held in RAM also.

*Hence an important design decision with mongo is on the amount of RAM.*

# How data is written Using Journaling

In this section we will briefly look at how write operations are performed using Journaling.

MongoDB disk writes are lazy i.e. if we receive 1,000 increments in one second for the object, it will only be written once. Physical writes occurs a couple of seconds after the operation.

We will now see how actually an update happens in a mongod.

In the MongoDB system mongod is the primary daemon process. It handles data requests, manages data format and performs background management operations.

So the disk has the data files and the Journal files.



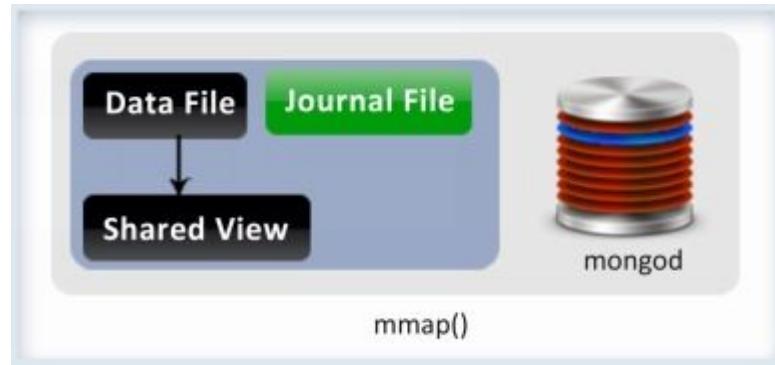
*Fig 7-39: mongod*

When the mongod is started the data files are mapped to a shared view.

That is the data files is mapped to a Virtual address space as explained above in the storage engine.

*So basically the OS recognizes that your data file is 2000 bytes on disk hence it maps this to memory address 1,000, 000 – 1,002, 000.*

*Also note as we have mentioned above the data will not be actually loaded until accessed, the OS just maps it and keeps it.*



**Fig 7-40: mmap to shared view**

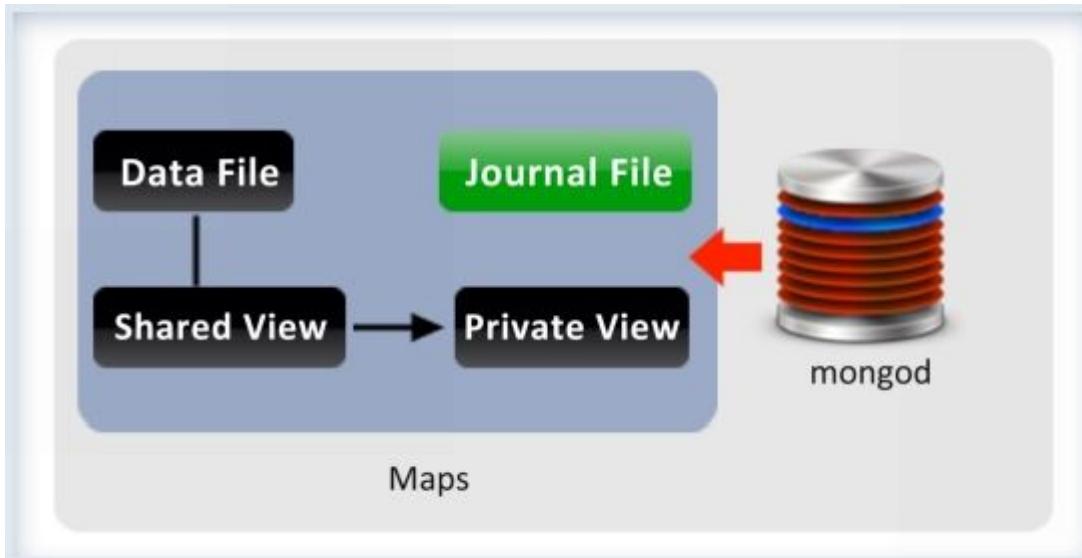
The memory is still backed by the file.

If any changes are made in the memory the OS will flush these changes to the underlying file.

*This is basically how the mongod works when journaling is not enabled – It asks the OS to flush in-memory changes every 60 seconds.*

In this scenario we are looking at writes with journaling enabled. Hence when journaling is enabled mongod makes a second mapping to a private view.

*This is why enabling journaling doubles the amount of virtual memory used by mongod.*

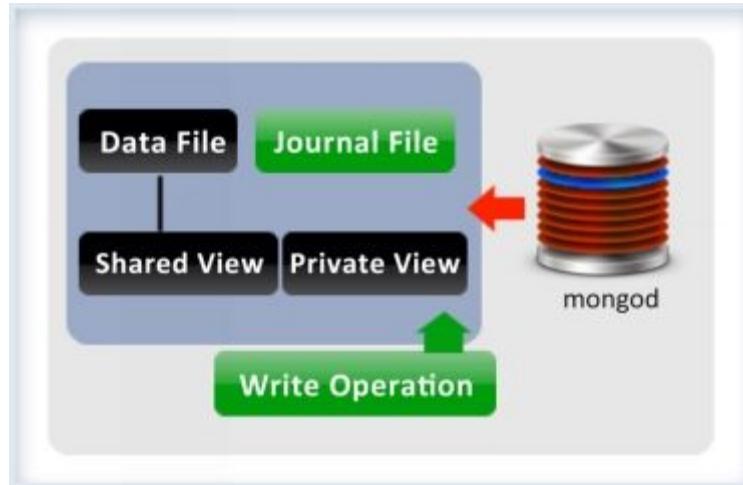


**Fig 7-41:** maps to private view

We can see in the above diagram the private view is not connected to the data file, hence the OS cannot flush any changes from the private view to the disk.

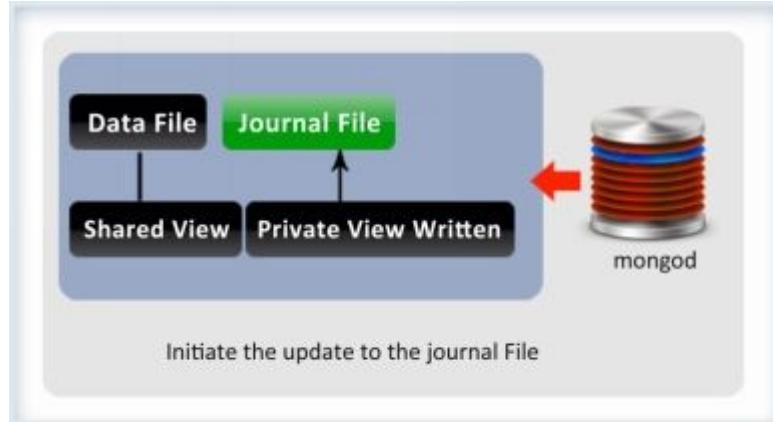
Let's see what sequence of events happens when a write operation is initiated.

When a write operation is initiated it first writes to the private view.



**Fig 7-42:** Initiated Write Operation

This will next write this change to the Journal file appending little description of which bytes in which file is changed.

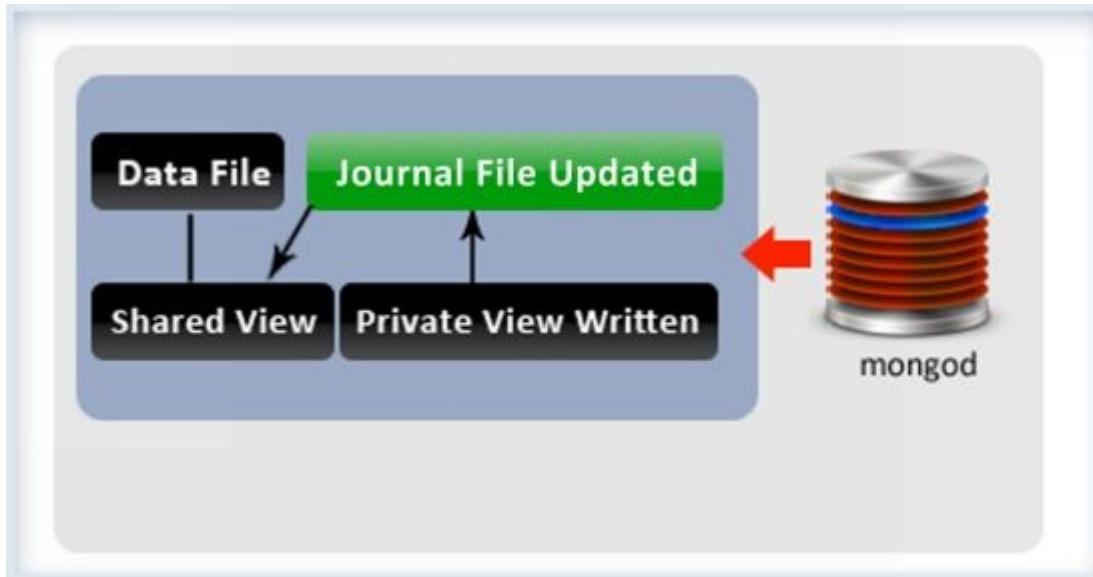


**Fig 7-43: Update Journal File**

The journal appends each change description as it gets.

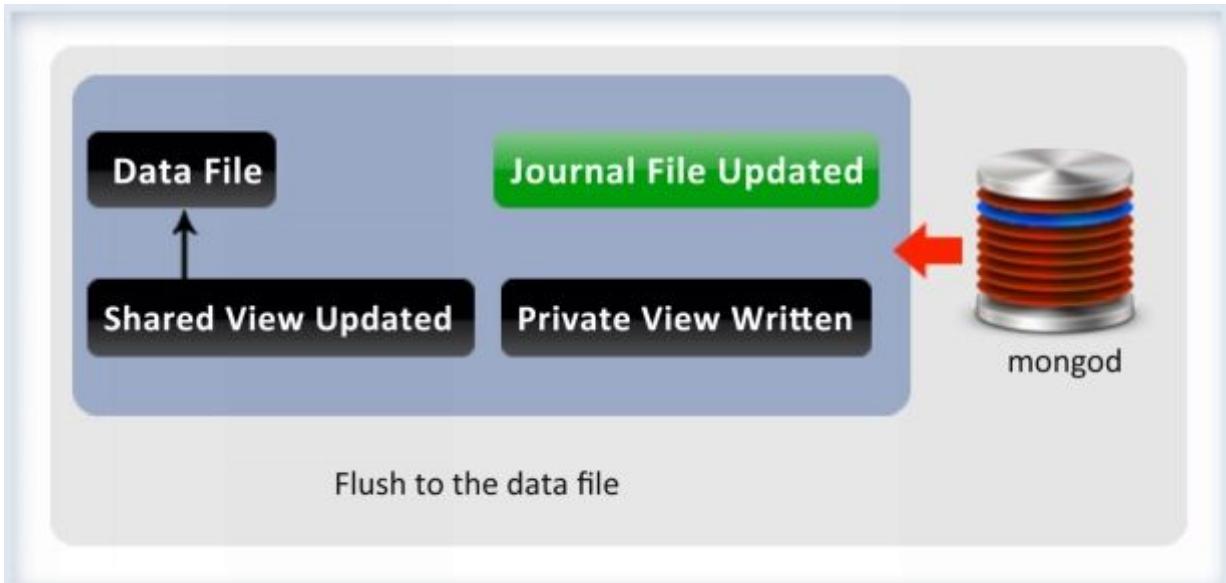
*At this point the write is safe. If mongod crashes the journal can replay the change even though the data file is not yet modified.*

The Journal will now replay the logged changes on the shared view.



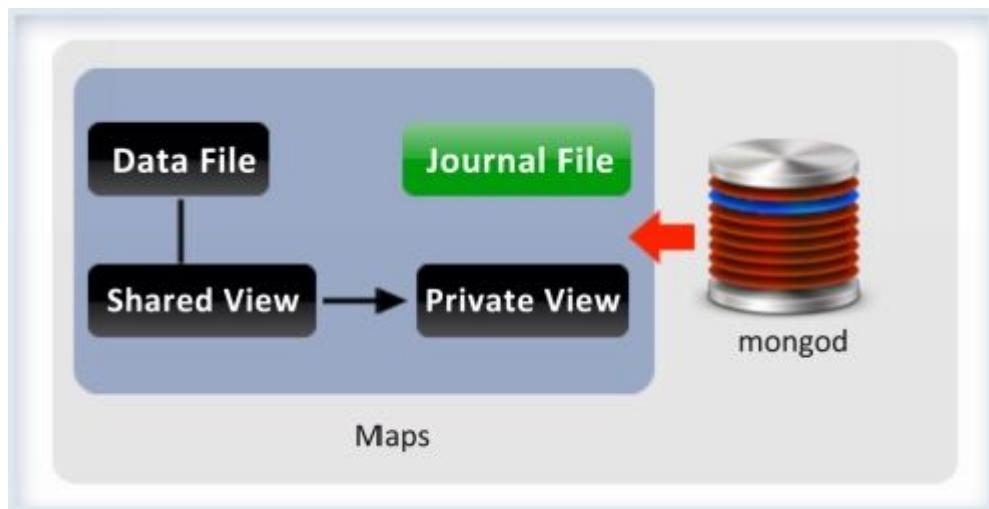
**Fig 7-44: Update Shared View**

Finally with a very fast speed the changes will be flushed to the disk. By default the mongod requests that the OS do this in every 60 seconds.



**Fig 7-45: Update Data File**

In the last step the mongod remaps the shared view to the private view. This prevents the private view from getting too dirty (having too many changes from the shared view it was mapped from).



**Fig 7-46: Re-Map**

## GridFS – The MongoDB File System

We have covered the architecture and we also looked at what happens under the hood. We have seen that MongoDB stores data in BSON documents. BSON documents have a document size limit of 16 MB. GridFS is a specification for handling large files in MongoDB that exceeds the document size limit of BSON. Hence in this section we will briefly understand about GridFS.

 *With Specification its means that it is not a MongoDB feature by itself i.e. there is no code in MongoDB that implements it, it just specifies how large files need to be handled and the language drivers such as PHP, Python etc. implement this specification and expose an API to the user of that driver enabling them to store/retrieve large files in MongoDB.*

## The rationale of GridFS

By design, a MongoDB document i.e. a BSON object cannot be larger than 16 megabytes. This is to keep performance at an optimum level and this could well be enough for your needs.

*For example, if you want to store a profile picture or a sound clip, then 4MB might be more space than you need.*

However if the requirement is to store movie clips, high quality audio clips or even files that are several hundred megabytes in size, under such scenario's MongoDB has you covered by using GridFS.

 *GridFS specifies a mechanism for dividing a large file among multiple documents.*

*The language driver that implements it, for example, the PHP driver, takes care of the splitting of the stored files (or merging the split chunks when files are to be retrieved) under the hood. The developer using the driver does not need to know of such internal details. This way GridFS allows the developer to store and manipulate files in a transparent and efficient way.*

GridFS works by storing the information about the file (called metadata) in one collection and the files data is broken down into pieces called chunks that are stored in another collection.

*In other words instead of storing a file in a single document, GridFS divides a file into parts, or chunks, and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k.*

This approach makes storing data both easy and scalable; it also makes range operations (such as retrieving specific parts of a file) much easier to use.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS.

You also can access information from arbitrary sections of files, which allows you to “skip” into the middle of a video or audio file.

*GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory.*

## GridFS under the hood

GridFS is a lightweight specification for storing files that is built on top of normal MongoDB documents.

 *The MongoDB server actually does almost nothing to “special case” the handling of GridFS requests; all of the work is handled by the client-side drivers and tools.*

The basic idea behind GridFS is that we can store large files by splitting them up into chunks and storing each chunk as a separate document.

Because MongoDB supports storing binary data in documents, we can keep storage overhead for chunks to a minimum.

In addition to storing each chunk of a file, we store a single document that groups the chunks together and contains metadata about the file.

GridFS actually stores the files using two collections named by default **fs.files** and **fs.chunks** although a different bucket name can be chosen than **fs**.

The chunks for GridFS are stored by default in the collection **fs.chunks**, but this can be overridden if needed.

*fs.chunks collection contains all the data of the user's files.*

Within the chunks collection the structure of the individual documents is pretty simple:

```
{  
  "_id" : ObjectId(...), "n" : 0, "data" :  
  BinData(...),  
  "files_id" : ObjectId(...)  
}
```

Like any other MongoDB document, the chunk has its own unique `"_id"`.

In addition, it has a couple of other keys.

- **"files\_id"** is the `"_id"` of the file document that contains the metadata for this chunk.
- **"n"** is the chunk number; this attribute tracks the order that chunks were present in the original file.
- Finally, **"data"** contains the binary data that makes up this chunk of the file.
- The metadata for each file is stored in a separate collection, which defaults to **fs.files**.
- Each document in the files collection represents a single file in GridFS and can contain any custom metadata that should be associated with that file.
- In addition to any user defined keys, there are a couple of keys that are mandated by the GridFS specification:
  - ***\_id - A unique id for the file—this is what will be stored in each chunk as the value for the "files\_id" key.***
  - ***Length - The total number of bytes making up the content of the file.***

- **chunkSize** - *The size of each chunk comprising the file, in bytes. The default is 256K, but this can be adjusted if needed.*
- **uploadDate** - *A timestamp representing when this file was stored in GridFS.*
- **md5** - *An md5 checksum of this file's contents, generated on the server side. Of all of the required keys, perhaps the most interesting (or least self-explanatory) is "md5". The value for "md5" is generated by the MongoDB server using the filemd5 command, which computes the md5 checksum of the uploaded chunks. This means that users can check the value of the "md5" key to ensure that a file was uploaded correctly.*

A typical fs.files document looks as follows

```
{  
  // unique ID for this file  
  "_id" : <unspecified>,  
  // size of the file in bytes  
  "length" : data_number,  
  // size of each of the chunks. Default is 256k  
  "chunkSize" : data_number,  
  // date when object first stored  
  "uploadDate" : data_date,  
  // result of running the "filemd5" command on  
  // this file's chunks  
  "md5" : data_string  
}
```

# Using GridFS

In this section we will be using the PyMongo driver to see how we can start using GridFS.

## Add Reference to the filesystem

The first thing which is needed is a reference to the GridFS filesystem

```
>>> import pymongo  
>>> import gridfs  
>>> conn=pymongo.Connection()  
>>> db=conn.gridfs_test  
>>> fs=gridfs.GridFS(db)  
write()
```

Next we will execute a basic write

```
>>> with fs.new_file() as fp:  
    fp.write('This is my new file. It is teh awezum!')
```

## find()

Let's examine using the mongo shell what the underlying collections holds

```
>>> list(db.fs.files.find())  
[{'_id': ObjectId('52fdd6189cd2fd08288d5f5c'),  
 '_length': 38, 'uploadDate':  
  datetime.datetime(2014, 2, 14, 8, 38, 48, 536000),  
 '_md5': 'u'332de5ca08b73218a8777da69293576a',  
 '_chunkSize': 262144}]  
>>> list(db.fs.chunks.find())  
[{'files_id': ObjectId('52fdd6189cd2fd08288d5f5c'),  
 '_id': ObjectId('52fdd6189cd2fd08288d5f5d'), 'data':  
  Binary('This is my new file. It is teh awezum!', 0), 'n':  
  0}]
```

You can see that there's really nothing surprising or mysterious happening there; it's just mapping the file system metaphor onto MongoDB documents.

## Force split the file

In this case, our file was small enough that it didn't need to be split into chunks. We can force split it by specifying a small chunkSize when creating the file:

```
>>> with fs.new_file(chunkSize=10) as fp:  
    fp.write('This is file number 2. It should be split into  
    several chunks')  
>>>  
>>> fp  
<gridfs.grid_file.GridIn object at 0x0000000002AC79B0>  
>>> fp._id  
ObjectId('52fdd76e9cd2fd08288d5f5e')  
>>> list(db.fs.chunks.find(dict(files_id=fp._id)))  
.....  
[ObjectId('52fdd76e9cd2fd08288d5f65'), u'data':  
Binary('s', 0), u'n': 6}]  
read()
```

We now know how the file is actually stored in the database; next using the client driver we will now read the file

```
>>> with fs.get(fp._id) as fp_read:  
    print fp_read.read()
```

This is file number 2. It should be split into several chunks

Hence we noticed that the user need not be aware of the chunks and all, you need to use the API's exposed by the client to read and write files from GridFS.

## Treating GridFS more Like a File System

There are several other convenience methods bundled into the GridFS object to give more filesystem like behavior.

## `new_file()`

For instance, `new_file()` takes any number of keyword arguments that will get added onto the `fs.files` document being created:

```
>>> with fs.new_file(  
    filename='file.txt',  
    content_type='text/plain',  
    my_other_attribute=42) as fp:  
    fp.write('New file')  
>>> fp  
<gridfs.grid_file.GridIn object at 0x0000000002AC7AC8>
```

```
>>> db.fs.files.find_one(dict(_id=fp._id))
{u'contentType': u'text/plain', u'chunkSize': 262144,
u'my_other_attribute': 42, u'filename': u'file.txt',
u'length': 8, u'uploadDate': datetime.datetime(2014, 2,
14, 8, 50, 35, 20000), u'_id':
ObjectId('52fdd8db9cd2fd08288d5f66'), u'md5':
u'681e10aecbaf7dd385fa51798ca0fd6'}
>>>
```

A file can also be overwritten using filenames. But since GridFS actually indexes files by \_id, it doesn't get rid of the old file, it just versions it:

```
>>> with fs.new_file(filename='file.txt',
content_type='text/plain') as fp:
    fp.write('Overwrite the so-called "New file"')
```

## **get\_version()/get\_last\_version()**

In the above case if we want to retrieve the file by filename, we can use get\_version or get\_last\_version:

```
>>> fs.get_last_version('file.txt').read()
'Overwrite the so-called "New file"'
>>> fs.get_version('file.txt',0).read()
'New file'
```

Since files have been uploaded with a filename property, we can also list the files in gridfs:

```
>>> fs.list()
[u'file.txt', u'file2.txt']
```

## **delete()**

Files can also be removed

```
>>> fp=fs.get_last_version('file.txt')
>>> fs.delete(fp._id)
>>> fs.list()
```

```
[u'file.txt', u'file2.txt']
>>> fs.get_last_version('file.txt').read()
'New file'
>>>
```

Note that since only one version of "file.txt" was removed, we still have a file named "file.txt" in the filesystem.

### **exists() and put()**

Finally, GridFS also provides convenience methods for determining if a file exists and for quickly writing a short file into GridFS:

```
>>> fs.exists(fp._id)
False
>>> fs.exists(filename='file.txt')
True
>>> fs.exists({'filename':'file.txt'}) # equivalent to
above
True
>>>     fs.put('The      quick      brown      fox',
filename='typingtest.txt')
ObjectId('52fddbc69cd2fd08288d5f6a')
>>> fs.get_last_version('typingtest.txt').read()
'The quick brown fox'
>>>
```

# Indexing

In this part of the book we will briefly understand what an index is in MongoDB context following that we will highlight the various types of indexes available in MongoDB, finally concluding the section by highlighting the behavior and limitations.

An index is a data structure that fastens up the read operations. In layman terms its comparable to a book index where we can reach out to any chapter by looking up the index for the Chapter Name and jumping directly to the page number rather than scanning the entire book to reach to the chapter which otherwise will be the case if no index is maintained.

Like in the book example where the Chapter Name is used for making the search better in databases also an index is defined on fields which can help in searching information in a better and efficient manner.

Like in other databases in MongoDB also it's perceived in the similar fashion i.e. it's used for speeding up the find () operation. The type of queries you run help in creating efficient indexes for the databases. For e.g. if most of the queries use Date field then it'll be beneficial to create an index on the Date field. It can be tricky to figure out what the optimal index for your queries is, but it is well worth the effort because Queries that otherwise take minutes can be instantaneous with the proper index in place.

In MongoDB an index can be created on any fields or sub-fields contained in the documents within a collection. Before we look at the various types of indexes which can be created in MongoDB we will list down few core features of the indexes.

MongoDB indexes have the following core features:

- The indexes are defined per-collection level. Hence for each collection we will have different sets of indexes.
- Like SQL Indexes a MongoDB index can also be created either on a single field or set of fields.
- Like we know in SQL that though indexes enhance the query performance but it incurs overhead for every write operations. So before creating any index we will need to consider the type of queries, its frequency, the size of the workload and also the insert load along with application requirements.
- All MongoDB indexes use a B-Tree data structure.
- Every query using the UPDATE operations uses only one index which is decided by the query optimizer. This can be overridden by using HINT method.
- An Index covers a query if all the fields in the query are part of the index and all the fields returned in the documents that match the query are also in the same index.
- When an index covers a query then the server can match the query conditions and return the results using the index only without any need to read the complete documents. Using queries with good index coverage reduces the number of full documents that MongoDB needs to store in memory, thus maximizing database performance and throughput.
- If an update does not change the size of a document or cause the document to outgrow its allocated area, then MongoDB will update an index only if the indexed fields have changed. This improves performance. Note that if the document has grown and must move, all index keys must then update.

# Type of Indexes

In this section we will look at the type of indexes available in MongoDB.

## \_id index

MongoDB by default creates the \_id index for all collections on \_id field. This index cannot be deleted.

## Secondary Indexes

All Indexes which are user created using `ensureIndex()` in MongoDB are termed as Secondary Indexes.

1. These indexes can be created on any field in the document or the sub document.

For e.g. let's consider a document

```
{"_id": ObjectId(...), "name": "John Doe",
"address": {"street": "Main", "zipcode": 53511,
"state": "WI"}}
```

In this document an index can be created on the “name” field as well as the “state” field.

2. These indexes can be created on a field which is holding a sub document.

If we consider the above document where “address” is holding a sub document, in that case an index can be created on the “address” field as well.

3. These indexes can either be created on a single field or set of fields. When created with set of fields it's also termed as **Compound indexes**.

*To explain it a bit further let's consider a collection names products that holds documents that resembles the following document*

```
{ "_id": ObjectId(...), "item": "Banana",
"category": ["food", "produce", "grocery"],
```

```
"location": "4th Street Store", "stock": 4, "type":  
cases, "arrival": Date(...)}
```

*If maximum of the queries use Item, Location and Stock fields then creating a compound index as below can help support those queries.*

```
db.products.ensureIndex ({"item": 1, "location":  
1, "stock": 1})
```

In addition to the query which are referring to all the fields of the compound index, the above compound index can also support queries which are using any prefix of the fields in the index i.e. it can also support queries which are using only item field or item and location field. However it will not support queries which select only the location field, only the stock field, only the location and stock fields or only the item and stock fields

4. If index is created on a field which hold an array as its value then MongoDB indexes each value in the array separately in a “multikey index”.

*For e.g. given the following document*

```
{ "_id" : ObjectId(...), "tags" : [ "weather",  
"hot", "record", "april" ] }
```

*Then an index on the tags field would be a multikey index and would include these separate entries:*

```
{ tags: "weather" }  
{ tags: "hot" }  
{ tags: "record" }  
{ tags: "april" }
```

5. Multikey compound indexes can also be created however at most only one field in a compound index can be of array type.

*For e.g. given a compound index on {a: 1, b: 1} the following documents are permissible:*

```
{a: [1, 2], b: 1}
```

`{a: 1, b: [1, 2]}`

However the following document is impermissible and also MongoDB cannot insert such a document into a collection with the `{a: 1, b: 1}` index:

`{a: [1, 2], b: [1, 2]}`

If an attempt is made to insert such a document, the insertion will be rejected and the following error results will be produced “cannot index parallel arrays”.

MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

We will next look at the various options/properties that might be useful while creating indexes .

## Indexes with Ascending and Descending Keys

An index maintains references to fields in either ascending or descending order.

When creating an index, the number associated with a key specifies the direction of the index. The options are: 1 (ascending) and -1 (descending). In a single key index it might not be that much important however the direction is very important in compound indexes because sometimes we might need the index fields running in opposite order relative to each other.

For e.g. Consider a collection of event data that includes both `username` and `timestamp`. And our query is to return events order by `Username first and then with the most recent event first`. Then in that case the following index will be used.

```
db.events.ensureIndex( { "username" : 1, "timestamp" : -1 } )
```

In the above example the index will contain references to documents sorted first by the values of the `username` field and, within each value of the `username` field, sorted by the values of `timestamp` field.

## Unique Indexes

At times when we are creating index we need to ensure uniqueness of the values being stored in the indexed field. In such cases we can create indexes with the “Unique” property true(by default it's false).

Say for e.g. we want to create a unique index on the “user\_id” field then we can issue the following command

```
db.addresses.ensureIndex( { "user_id": 1 }, {  
unique: true } )
```

This command ensures that we have unique values in the user\_id field. Few points which we need to note for the uniqueness constraint are

1. If the unique constraint is used on a compound index in that scenario uniqueness is enforced on the combination of values.
2. If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document
3. MongoDB will only permit one document without a unique value in the collection because of this unique constraint .

## dropDups

If we are creating a unique index on a collection which already has documents, then the creation might fail as we may have some documents which contain duplicate values in the indexed field. In such scenarios to force the creation of the unique index the “dropDups” options can be used. It works by keeping first occurrence of a value for the key, and delete all subsequent values. By default dropDups is false.

## Sparse Indexes

A Sparse index is an index which holds entries of the documents within a collection which has the fields on which the index is created. For e.g. if we want to create a sparse index on the “Last\_name” field in the “User” collection then the following command can be issued.

```
db.Users.ensureIndex( { "Last_Name": 1 }, {  
    sparse: true } )
```

This index will contain documents such as

```
{FirstName: Test, LastName: User}
```

or

```
{FirstName: Test2, LastName: }
```

However the following document will not be part of the sparse index.

```
{FirstName: Test1}
```

The index is sparse because of the missing documents when fields are missing however sparse indexes provides a significant space saving.

In contrast the non-sparse index will include all documents in a collection and will store null values for document that do not contain the indexed field

## TTL Indexes (Time To Live)

A new index property was introduced in the version 2.2 which enabled us an user to automatically remove documents from the collection after a specified time period. This is ideal for scenarios such as machine generated event data, logs, and session information which need to persist in a database for a limited amount of time. Say for e.g. if we want to set the TTL of one hour on collection “logs” the following command can be used

```
db.Logs.ensureIndex( { "Sample_Time": 1 }, {  
    expireAfterSeconds: 3600 } )
```

However we need to note the following limitations these indexes have

1. *The indexed field must be a date type i.e. in our above example the field “sample\_time” must hold date values.*
2. *Compound indexes are not supported*
3. *If the field holds an array, and there are multiple date-typed data in the index, the document will expire when the lowest*

- (i.e. earliest) matches the expiration threshold.
4. This cannot be created on a field that already has an index created.
  5. This index cannot be created on capped collections.
  6. TTL indexes expires data by removing documents in a background task that runs once a minute. As a result, the TTL index provides no guarantees that expired documents will not exist in the collection.

## Geospatial Indexes

With the emergence of mobile devices the query of finding things nearing a current location is becoming very common. MongoDB provides “geospatial indexes” to support location-based and other similar queries in a two dimensional coordinate systems.

To create a geospatial index the documents must have a coordinate pair in the form of a two element array or embedded documents with two keys.



The following are valid examples

```
{ "loc" : [ 0, 100 ] }  
{ "loc" : { "x" : -30, "y" : 30 } }  
{ "loc" : { "latitude" : -180, "longitude" : 180 } }
```

The keys can be anything; for example, `{"loc" : {"foo" : 0, "bar" : 1}}` is valid.

Geospatial indexes can be created on the field `loc` in the following way

```
db.places.ensureIndex( { loc : "2d" } )
```

By default, geospatial indexing assumes that your values are going to range from -180 to 180 (which are convenient for latitude and longitude).

*The maximum and minimum values can be specified as options to ensureIndex.*

```
db.star.trek.ensureIndex({"light-years" : "2d"},  
 {"min" : -1000, "max" : 1000})
```

*This will create a spatial index calibrated for a 2,000 × 2,000 light-year square.*

Any documents that have values in the field beyond the maximum and the minimum values will be rejected.

We can also create compound Geospatial indexes.

*For e.g. if we have documents which are of the type as follows*

```
{"loc": [0, 100], "desc": "coffeeshop"}  
 {"loc": [0, 1], "desc": "pizzashop"}
```

*And the query of a user is to find out all coffee shops near to its location then the following compound index can quickly find the nearest coffee shop*

```
db.ensureIndex({"location" : "2d", "desc" : 1})
```

## Geohaystack Indexes

In addition to conventional geospatial indexes MongoDB also provides a bucket-based geospatial index, called “**geospatial haystack indexes**.”

These indexes support high performance queries for locations within a small area, when the query must filter along another dimension.



*Example: If you need to return all documents that have coordinates within 25 miles of a given point and have a type field value of “museum,” a haystack index would provide the best support for these queries.*

Haystack indexes allow you to tune your bucket size to the distribution of your data, so that in general you search only very small regions of 2d space for a particular kind of document.

*These indexes are not suited for finding the closest documents to a particular location, when the closest documents are far away*

*compared to bucket size.*

The `bucketSize` parameter is required, and determines the granularity of the haystack index.

For example:

```
db.places.ensureIndex({ pos : "geoHaystack",
    type : 1 }, { bucketSize : 1 })
```

*This example bucketSize of 1 creates an index where keys within 1 unit of longitude or latitude are stored in the same bucket. An additional category can also be included in the index, which means that information will be looked up at the same time as finding the location details.*

*If your use case typically searches for "nearby" locations (i.e. "restaurants within 25 miles") a haystack index can be more efficient.*

*The matches for the additional indexed field (e.g. category) can be found and counted within each bucket.*

*If, instead, you are searching for "nearest restaurant" and would like to return results regardless of distance, a normal 2d index will be more efficient.*

There are currently (as of MongoDB 2.2.0) a few limitations on haystack indexes:

1. only one additional field can be included in the haystack index
2. the additional index field has to be a single value, not an array
3. null long/lat values are not supported

In addition to the above mentioned type we have a new type of index introduced in version 2.4 which supports text search on a collection.

## Behaviors are Limitations

Finally we will conclude the index section by listing down few behaviors and limitations which as an user you need to be aware of.

1. A collection may have no more than [64 indexes](#).
2. Index keys can be no larger than [1024 bytes](#).

3. Documents with fields that have values greater than this size cannot be indexed.
4. To query for documents that were too large to index, you can use a command similar to the following:
 

```
db.myCollection.find({<key>: <value too large to index>}).hint({$natural: 1})
```
5. The name of an index, including the [namespace](#) must be shorter than [128 characters](#).
6. Indexes have storage requirements, and impacts insert/update speed to some degree.
7. Create indexes to support queries and other operations, but do not maintain indexes that your MongoDB instance cannot or will not use.
8. For queries with the [\\$or](#) operator, each clause of an [\\$or](#) query executes in parallel, and can each use a different index.
9. For queries that use the [sort\(\)](#) method and use the [\\$or](#) operator, the query cannot use the indexes on the [\\$or](#) fields.
10. 2d [geospatial queries](#) do not support queries that use the [\\$or](#) operator.

## Summary

In this chapter we covered the Core Processes and Tools, Standalone deployment, Sharding concepts, Replication concepts and Production Deployment and we also looked at how HA can be achieved. We have seen how data is stored under the hood and how writes happens using Journaling, concluding the chapter with GridFS, different types of indexes available in MongoDB.

In the following chapter we will cover MongoDB from administration perspective.

# Administering MongoDB

*“Administering a MongoDB is not like administering the traditional RDBMS databases. Though most of the administrative tasks are not required or is quick and done automatically by the system, however there are some administrative tasks which require manual intervention.”*

In this chapter we will go over the process of basic administrative operations for: Backups and Restoration, Importing and Exporting data, Managing Server and monitoring the database instances

# Administration Tools

The first thing is awareness of the tools which can be used to perform the administrative operations on MongoDB. Hence before we dive into the administration tasks we will have a quick overview of what are the tools.

Since MongoDB does not have a GUI-style administrative interface hence most of the administrative tasks are done using the command line mongo shell, however some UI's are available as separate community projects.

Hence for administration purposes in MongoDB mostly the following tools are used

1. mongo shell and
2. 3<sup>rd</sup> party tools

## mongo

The mongo shell is an interactive JavaScript shell for MongoDB and is part of the MongoDB distributions.

It provides a powerful interface for administrators as well as developers to test queries and operations directly with the database.

In the previous chapters we have already covered the development using the shell. In this chapter we will go through the system administration tasks using the shell.

## Third Party Administration Tool

A number of third party tools are available for MongoDB. Most of the tools are web based.



*List of the entire third party administration tool supporting MongoDB is maintained by 10gen on the [MongoDB web site](http://docs.mongodb.org/ecosystem/tools/administration-interfaces/) (<http://docs.mongodb.org/ecosystem/tools/administration-interfaces/>)*

# Backup and Recovery

Backup is one of the most important administrative tasks which ensure that the data is safe and in case of any emergency can be restored back.

If the data cannot be restored back then the backup is useless, hence after taking a backup the administrators need to ensure that it's in a usable format and has captured data in a consistent state.

Hence the first skills that administrators need to learn is taking backups and restoring it back.

## Data File Backup

The easiest way of backing up the database is to copy the data in the data directory folder.

 *MongoDB stores all of its data in a data directory. By default, this directory is /data/db/ (or C:\data\db\ on Windows) but this can be configured to point to different directory using –dbpath option while starting MongoDB.*

The contents of the data directory form a complete representation of the data stored in MongoDB.

Hence taking backup of MongoDB is as simple as creating a copy of all of the files in the data directory.

It is general not safe to copy the data directory while MongoDB is running, hence one option for taking the backup is to shut down the MongoDB server and then copy the data directory.

Assuming the server is shut down safely, the data directory will represent a safe snapshot of the data stored when it was shut down. That directory can be copied as a backup before restarting the server.

Although shutting down the server and copying the data directory is an effective and safe method of taking backups, it is not ideal. Hence

we'll look at techniques for backing up MongoDB without requiring any downtime.

## **mongodump and mongorestore**

### ***mongodump***

mongodump is the MongoDB backup utility which is supplied as part of the MongoDB distributions. mongodump works by querying against a running MongoDB server and writing all of the documents it contains to disk. mongodump is just a regular client hence it can be run against a live instance of MongoDB, even the one which is handling other requests and performing writes.

Let's begin by performing a simple backup and then restore it to validate that the backup is in usable and consistent format.

The following code snippets are from running the utilities on a windows platform. For the example which follows the MongoDB server is running on the localhost instance. Follow the following command to start the mongod.

*Open a terminal window and enter the following command.*

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongod
```

*mongod --help for help and startup options*

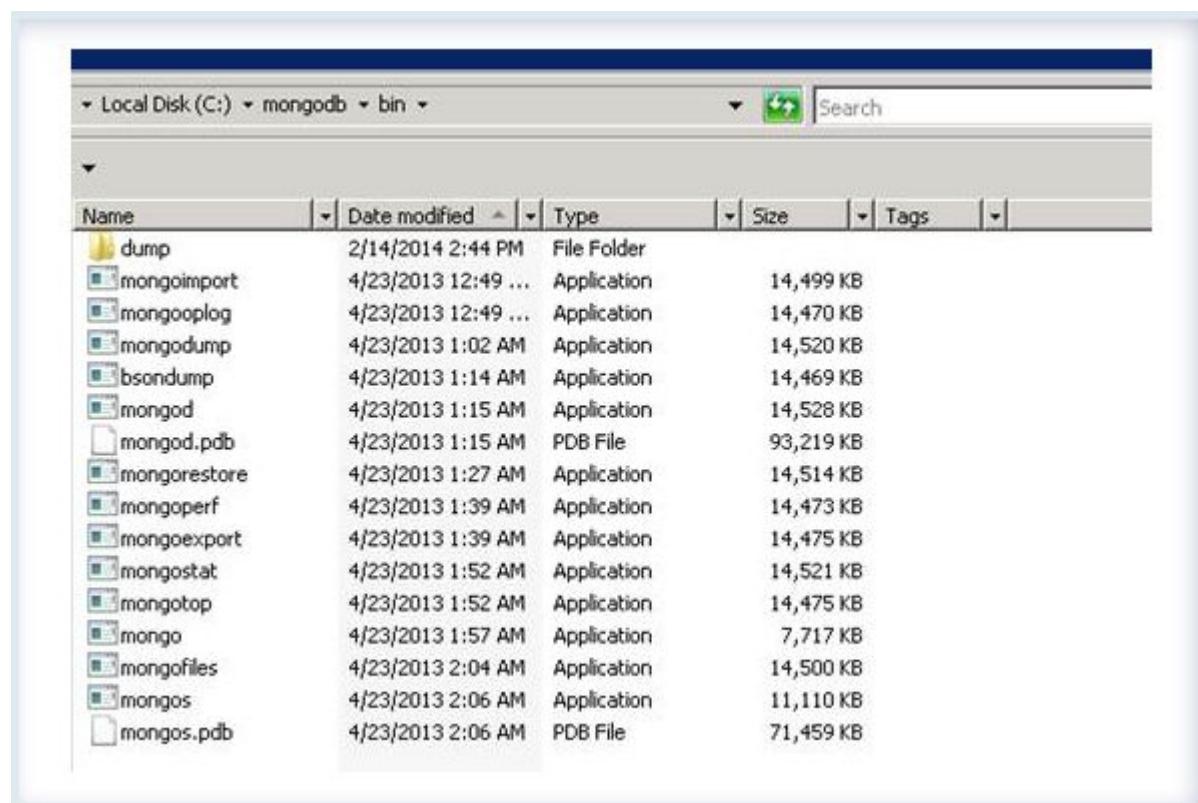
```
.....  
Fri Feb 14 14:38:30.387 [initandlisten] waiting for connections on port 27017
```

```
Fri Feb 14 14:38:30.387 [websvr] admin web console  
waiting for connections on port 28017
```

In order to run mongodump, open a new terminal window and execute the following

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongodump  
connected to: 127.0.0.1  
Fri Feb 14 14:43:55.913 all dbs  
.....  
Fri Feb 14 14:44:30.973 Metadata for  
products.system.users to dump\produ  
cts\system.users.metadata.json  
c:\mongodb\bin>
```

This dumps the entire database under the dump folder in the bin folder directory itself.



*Fig 8-1: dump folder*

The mongodump utility by default connects to the localhost interface MongoDB database using the default port. Next it pulls each database and collections associated data files and store them in a predefined folder structure which by default is **`./dump/[databasename]/[collectionname].bson`**.

The data is saved in .bson format which are similar to the format that MongoDB uses internally to store its data.

*If existing contents are there in the directory it will remain untouched unless the dump contains same file, for e.g. if the dump contains the file c1.bson and c2.bson and the output directory has file c3.bson and c1.bson, then mongodump will replace the c1.bson file of the folder with its c1.bson file, will copy the c2.bson file but it'll not remove/change the c3.bson file.*

You should ensure that the directories content is deleted before the same is used for mongodump unless you have requirement of overlaying the data in your backups.

## **Single Database Backup**

In the above example we executed mongodump with the default setting which dumps all the databases on the MongoDB database server.

In the actual scenario where you will have multiple applications running on the same server with different databases the requirement might be to set up different backup strategies for each database. This can be achieved by specifying –d parameter to the command line.

Specifying –d parameter in mongodump utility will enable the users to take backup's database wise.

```
c:\mongodb\bin>mongodump -d mydbpoc
connected to: 127.0.0.1
Fri Feb 14 14:51:56.388 DATABASE: mydbpoc to
dump\mydbpoc
.....
```

```
Fri Feb 14 14:51:57.277 Metadata for  
mydbpoc.MR_ClassAvg_1  
to  
dump\mydbpoc\MR_ClassAvg_1.metadata.json  
c:\mongodb\bin>
```

## **Collection Level Backup**

We have two types of data in every database one which changes rarely such as configuration data where we maintains the users, their roles, any application related configurations and then we have data which changes frequently such as the Events data(in case of Monitoring Application), Posts data (in case of Blog application) and so on.

Hence the backup requirement of both this type of data is different say for instance the complete database can be backed up once a week whereas the rapidly changing collection need to backed up every hour.

Specifying –c parameter in mongodump utility enables the users to implement backups for specified collection individually.

```
c:\mongodb\bin>mongodump -d mydbpoc -c users  
connected to: 127.0.0.1
```

```
-----  
Fri Feb 14 14:53:31.317 Metadata for mydbpoc.users to  
dump\mydbpoc\users  
.metadata.json  
c:\mongodb\bin>
```

 *Note: if the folder where the data need to be dumped is not specified then by default it dumps the data in a directory names dump in the current working directory which in this case is c:\mongodb\bin.*

## ***[mongodump –help](#)***

We have covered the basics of executing mongodump apart from the options mentioned above mongodump provides various options enabling the users to tailor the backups as per the requirement.

As with all the other utility executing the utility with –help option will provide the list of all available options.

## ***mongorestore***

As we mentioned above its mandatory for the administrators to ensure that the backups which are happening is in consistent and usable format. Hence the next step is to restore the data dump back using mongorestore.

This utility when run will restore the database back to the state of time when the dump was taken.

```
c:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongorestore  
connected to: 127.0.0.1
```

```
.....  
Fri Feb 14 15:14:10.112 Creating index: { key: { user: 1,  
userSource: 1  
}, unique: true, ns: "products.system.users", name:  
"user_1_userSource_1" }
```

```
c:\mongodb\bin>
```

*This will force the data to be appended to the end of existing data.*

*To override the default behavior –drop should be used in the above snippet.*

*The –drop indicates the mongorestore utility that it needs to delete all the collections and data within the aforementioned database and then restore the dump data back to the database.*

*If –drop is not used then the data will be appended to the end of the existing data.*

## ***Restoring Single database***

As we have seen in the backup section, the backup strategies can be specified at individual database level i.e. we can run mongodump to take backup of a single database by using –d option.

Similar to that we can specify –d option to mongorestore to restore individual databases.

```
c:\mongodb\bin>mongorestore -d mydbpoc  
C:\mongodb\bin\dump\mydbpoc --drop  
connected to: 127.0.0.1.....  
Fri Feb 14 15:17:52.006 Creating index: { key: { _id: 1 },  
ns: "mydbpoc.  
users", name: "_id_" }  
c:\mongodb\bin>
```

## ***Restoring Single collection***

As with mongodump where we can use –c option to specify collection level backups, similar to that we can also restore individual collections by using –c option with mongorestore utility.

```
c:\mongodb\bin>mongorestore -d mydbpoc -c users  
C:\mongodb\bin\dump\mydbpoc\users.bson --drop  
connected to: 127.0.0.1.....
```

## 22 objects found

```
Fri Feb 14 15:19:27.083 Creating index: { key: { _id: 1 },  
ns: "mydbpoc.  
users", name: "_id_" }  
c:\mongodb\bin>
```

### **Mongorestore –help**

The mongorestore also have multiple options. The same can be viewed using –help option.



<http://docs.mongodb.org/manual/core/backups/>

## **fsync and Lock**

Although mongodump and mongorestore allow us to take backups without shutting down the MongoDB server, we lose the ability to get a point-in-time view of the data.

We have also seen how we can copy the data files to take the backups but for copying that we need a downtime where we will need to shut down the server before copying the data which is not feasible in a production environment.

MongoDB's fsync command allows us to copy the data directory of a running MongoDB server without risking any corruption.

The fsync command will force the MongoDB server to flush all pending writes to disk. It will also, optionally, hold a lock preventing any further writes to the database until the server is unlocked. This write lock is what allows the fsync command to be useful for backups.

We will next see how to run the command from the shell, forcing an fsync and acquiring a write lock. In order to do so, open a terminal window and connect to the mongo console.

```
c:\mongodb\bin>mongo  
MongoDB shell version: 2.4.9
```

*connecting to: test*

>

Next switch to admin db and issue the runCommand to fsync

> `use admin`

*switched to db admin*

> `db.runCommand({ "fsync": 1, "lock": 1 })`

{

`"info" : "now locked against writes, use db.fsyncUnlock() to unlock",`

`"seeAlso" :`

`"http://dochub.mongodb.org/core/fsynccommand",`

`"ok" : 1`

}

>

At this point, the data directory represents a consistent, point-in-time snapshot of our data because the server is locked for writes hence we can safely make a copy of the data directory to use as a backup.

After performing the backup, we need to unlock the database again.

In order to do issue the following command

> `db.$cmd.sys.unlock.findOne()`

{ `"ok" : 1, "info" : "unlock completed"` }

>

*The currentOp command can be used to ensure that the lock has been released.*

> `db.currentOp()`

{ `"inprog" : [ ]` }

*(It may take a moment after the unlock is first requested.)*

The fsync command allows us to take very flexible backups, without shutting down the server or sacrificing the point-in-time nature of the backup.

The price we've paid, however, is a momentary block against write operations.

The only way to have a point in-time snapshot without any downtime for reads or writes is to backup from a slave.

## Slave Backups

Although the options discussed earlier provide a wide range of flexibility in terms of backups, none is as flexible as backing up from a slave server.

When running MongoDB with replication, any of the previously mentioned backup techniques can be applied to a slave server rather than the master.

The slave will always have a copy of the data that is nearly in sync with the master. Because we're not depending on the performance of the slave or its availability for reads or writes, we are free to use any of the three options above: shutting down, the dump and restore tools, or the `fsync` command.

*Backing up from a slave is the recommended way to handle data backups with MongoDB.*

# Importing and Exporting

When we are trying to migrate our application from one environment to another then often is the case to import data or to export data.

## mongoimport

MongoDB provides mongoimport utility which enables the users to bulk load data directly into a collection of the database.

It reads from a file and bulk loads the data into a collection.

The following three file formats are supported by mongoimport

1. CSV – This is a comma separated file.
2. TSV – This is similar as a CSV file the only difference is it uses TAB as the separator.
3. JSON – This contains one block of JSON per line that represents a document.

Using –help with mongoimport will provide all the options available with the utility.

The utility of the mongoimport is very simple.

Most of the time we will end up using the following options

**-h or --host:** Specifies a resolvable hostname for the mongod to which you want to restore the database.

*By default mongoimport will attempt to connect to a MongoDB process running on the localhost port numbered 27017.*

Optionally, specify a port number to connect a MongoDB instance running on a port other than 27017.

**-d or --db:** Specify the database where the data need to be imported

**-c or --collection:** Specifies the collection where data need to be uploaded

**--type:** this is either of the file type i.e. CSV, TSV or JSON.

**--file:** this is the file path from where the data need to be imported

**--drop:** If this is not set then the data will be appended to the collection otherwise this option drops the collection and recreates it from the imported data.

**--headerLine:** This is used for CSV or TSV files only, is used to indicate that the first line is header line.

The following command import the data from a csv file to the testimport collection on the localhost.

```
c:\mongodb\bin>mongoimport --host localhost --db mydbpoc --collection testimport --type csv --file c:\exporteg.csv --headerline  
connected to: localhost  
Fri Feb 14 15:44:43.506 check 9 16  
Fri Feb 14 15:44:43.507 imported 15 objects  
c:\mongodb\bin>
```

## mongoexport

Similar to mongoimport utility MongoDB provides a mongoexport utility which enables the user to export data from the MongoDB database.

As the name implies it creates export files from existing MongoDB collections.

Using –help provides available options with the mongoexport utility:

Prominent options which we as a user will end up utilizing most of the time are

**-q:** This is used to specify the query that will return as output the records that need to be exported. This is similar to what we specify in the db.CollectionName.find() function when we have to retrieve records matching the selection criteria. If no query is specified all the documents are exported.

**-f:** This is used to specify the fields that we need to export from the selected documents.

The following command exports the data from Users collection to a CSV file.

```
c:\mongodb\bin>mongoexport -d mydbpoc -c users -f _id,Age --csv > users.csv  
connected to: 127.0.0.1  
exported 22 records
```

```
c:\mongodb\bin>
```



<http://docs.mongodb.org/manual/reference/program/mongoimport/>

<http://docs.mongodb.org/manual/reference/program/mongoexport/>

## Managing the Server

In this section we will look at various options which we need to be aware of as an administrator of the system.

### Start a Server

In this section we will briefly understand how we can start the server.

In most part of the book we have already used the mongo shell to start the server by running mongod exe.

The MongoDB server can be started manually by opening a command prompt (run as administrator) in windows or a terminal window on LINUX systems and type the following command.

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongod  
mongod --help for help and startup options
```

.....

*This window will display all the connections which is being made to the mongod. It also displays information which can be used to monitor the server working.*

If no configuration is specified the MongoDB starts up with the default database path of /data/db (on Linux) and C:/Data/Db(on

Windows) and binds to the localhost(127.0.0.1) using default ports 27017 and 27018.

*Typing ^C causes the server to shut down cleanly.*

MongoDB provides two methods for specifying configuration parameters for starting up the server.

First is to specify using command-line options in conjunction with the mongod server daemon (refer Installation and configuration chapter).

Second method is to load a configuration file. The server configuration can be changed by editing the file and then restarting the server.

# Stop a Server

The server can be shut down pressing CTRL+C in the mongod console itself otherwise we can use shutdownServer command from the mongo console.

Open a terminal window, connect to the mongo console.

```
C:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongo  
MongoDB shell version: 2.4.9  
connecting to: test  
>
```

Switch to admin db and issue the shutdownServer command.

```
> use admin  
switched to db admin  
> db.shutdownServer()  
Fri Feb 14 15:53:57.679 DBCClientCursor::init call() failed  
server should be down...  
Fri Feb 14 15:53:57.683 trying reconnect to  
127.0.0.1:27017  
Fri Feb 14 15:53:58.677 reconnect 127.0.0.1:27017  
failed couldn't connect to ser  
ver 127.0.0.1:27017  
>
```

*If you will check the mongod console where you started the server in the previous step, you will see the server has been shut down successfully.*

```
.....  
Fri Feb 14 15:53:57.632 [conn1] shutdown: removing fs  
lock...  
Fri Feb 14 15:53:57.632 dbexit: really exiting now  
c:\mongodb\bin>
```

# View Log Files

MongoDB writes its entire log output to stdout by default; however it can be changed by specifying the logpath option in the configuration while starting the server to redirect the log output to a file instead.

The contents of the log file can be used to spot problems such as excessive connections and asserts (exceptions) that may indicate problems with the data.

## Server Status

db.ServerStatus() is a simple method provided by MongoDB for determining the status of the server such as server uptime, number of connections and so on.

In order to check the status, connect to the mongo console, switch to admin db and issue the db.serverStatus() command.

```
c:\mongodb\bin>mongo  
MongoDB shell version: 2.4.9  
connecting to: test  
> use admin  
switched to db admin  
> db.serverStatus()
```

"opcounters" and "asserts" section provides useful information which can be analyzed for classifying likely problem if any.

The "opcounters" section shows the number of operations of each type that have been performed against the database server.

*In order to find out if there's any problem we should have a baselining of these operations, if the counters start deviating out of the baselining then it indicates a problem and will require to take actions to bring it back to the normal state.*

The "asserts" section shows the number of server and client exceptions or warnings that have been thrown. If such exceptions or

warnings start to rise rapidly, then it's time to take a good look through your server's logfiles to see whether a problem is developing. A high number of asserts may also indicate there is a problem with the data in the database, and using MongoDB's validation functions should be considered to check that the data is undamaged.

## Identify and Repair Server

In this section we will look at how we can repair a corrupt database.

If we are getting errors as below

1. Database server refuses to start stating data files are corrupt
2. Asserts are seen in the log files or db.serverStatus() command
3. Strange or unexpected queries results

This means the database is corrupted and repair must be run in order to recover the database.

The first thing which we need to do before we can start the repair is to shut down the server if it's not already offline. We can use either option mentioned above. As of now we typed ^C in the mongod console. This will shut down the server.

Next we will start the mongod using –repair option as shown below

```
c:\mongodb\bin>mongod --repair
```

```
Fri Feb 14 16:09:30.358 [initandlisten] MongoDB starting  
: pid=7028 port=27017 dbpath=\data\db\ 64-bit host=ANOC9
```

```
Fri Feb 14 16:09:30.359 [initandlisten] db version v2.4.9
```

---

```
Fri Feb 14 16:10:04.378 dbexit: really exiting now
```

```
c:\mongodb\bin>
```

This will repair the mongod, if you will look at the output you'll find various discrepancies that the utility is repairing. Once the repair process is over it exits.

Post the completion of the repair process the server can be started as normal and then the latest database backups can be used to restore missing data.

*At times when a large database is under repair we may find that the drive runs out of disk space. This happens because MongoDB may need to make a temporary copy of the database files on the same drive as the data.*

*In order to overcome this issue the MongoDB repair utility supports an additional command-line parameter called –repairpath which can be used to specify a drive with enough space to hold the temporary files it creates during the rebuild process.*

## Identify and Repair collection level data

It might happen at times we just want to validate that the collection hold valid data and had valid indexes. For such cases MongoDB provides a Validate() method which validates the content of the specified collection.

The following example validates the data of the “Users” collection

```
c:\mongodb\bin>mongo
MongoDB shell version: 2.4.9
connecting to: test
> use mydbpoc
switched to db mydbpoc
> db.users.validate()
{
  "ns" : "mydbpoc.users",
  "firstExtent" : "1:4322000 ns:mydbpoc.users",
  "lastExtent" : "1:4322000 ns:mydbpoc.users",
  ".....",
  "valid" : true,
  "errors" : [ ],
```

```
        "warning" : "Some checks omitted for speed. use  
{full:true} option to do  
more thorough scan.",  
        "ok" : 1  
    }
```

*By default the validate() option checks both the data files and the associate indexes and provides statistics of the collection as we can see in the above screenshot which will enable the users in identifying if there's any problem with the data files or the indexes.*

If running Validation () indicates that the indexes are damaged in that case reIndex can be used to re index the indexes of the collection. This will drop all the indexes on the collection and will rebuild the indexes.

The following command reindexes the Users collections' indexes.

```
> use mydbpoc
switched to db mydbpoc
> db.users.reIndex()
{
  "nIndexesWas" : 1,
  "msg" : "indexes dropped for collection",
  "nIndexes" : 1,
  "indexes" : [
    {
      "key" : {
        "_id" : 1
      },
      "ns" : "mydbpoc.users",
      "name" : "_id_"
    }
  ],
  "ok" : 1
}
>
```

If the collections data files are corrupted then running –repair option will be the best way for repairing all the data files.

# Monitoring MongoDB

As an administrator of the MongoDB server, it's important to monitor the health and performance of the system. In this section we will discuss ways of monitoring the system.

## mongostat

mongostat comes as part of the MongoDB distribution. This tool provides simple stats of the server, though it's not extensive but it provides a good overview. The following shows the statistics of the localhost. Open a terminal window and execute the following

```
c:\>cd c:\mongodb\bin  
c:\mongodb\bin>mongostat  
connected to: 127.0.0.1  
insert query update delete getmore command flushes  
mapped vsize res faults  
locked db idx miss % qr|qw ar|aw netIn netOut conn time  
*0 *0 *0 *0 0 1|0 0 288m 728m 43m 2  
local:0.0% 0 0|0 0|0 62b 2k 1 16:20:32  
*0 *0 *0 *0 0 1|0 0 288m 728m 43m 0  
mydbpoc:0.0% 0 0|0 0|0 62b 2k 1 16:20:33  
*0 *0 *0 *0 0 1|0 0 288m 728m 44m 31  
.....
```

The first six columns show the rate at which various operations are handled by the mongod server. Apart from this columns the following columns are also worth mentioning and can be of use when diagnosing problems

1. idx\_miss – This displays the percentage of query that could not use an index. High value indicates that we might need to consider adding indexes.

2. Conn – This is an indicator of the number of connections to the mongod instance. A high value here can indicate a possibility that the connections are not getting released or closed from the application which means that though the application is issuing an open connection it's not closing the connection after completion of the operation.
3. % locked – This shows the amount of time the collections were locked. High indicator means that blocking operations are being performed which can be best considered to be run during off-peak hours.

## **mongod web interface**

We have seen in the installation chapter whenever we start up the mongod by default it creates a web interface on the port number that is 1000 higher than the port number where it listens for the connection. By default the HTTP port is 28017.

Most of the statistical information is also available on this mongod web interface. This is accessed through your web browser. If we are running mongod on localhost which is listening for connections on 27017 port then in that case the HTTP status page can be accessed using the following URL: <http://localhost:28017>.

**Fig 8-2: Web interface**

## Third-Party Plug-Ins

In addition to this tool there are also many third-party adapters available for MongoDB that let you use common open source or commercial monitoring systems such as cacti, Ganglia etc. As mentioned previously, 10gen maintains a page on its website that shares the latest information about monitoring interfaces available for MongoDB.



For an up-to-date list of third-party plug-ins, check out the [www.mongodb.org/display/DOCS/Monitoring+and+Diagnostic](http://www.mongodb.org/display/DOCS/Monitoring+and+Diagnostic)

# Summary

In this chapter we have seen how we can use the various utilities which are packaged as part of the MongoDB distribution to manage and maintain the system.

We have covered main operations that as an administrator you must be aware of for detailed understanding of the utilities please delve through the references. In the next chapter we will cover MongoDB's use cases and we will also look at the NA ranges.

# MongoDB Use Cases

*“MongoDB - is it useful for me or is it not? This is what we will be talking about in this chapter.”*

In this chapter we will provide the much needed connection between the features of MongoDB and the business problems that it is suited to solve. We will be using two use cases for sharing the techniques and patterns used for addressing such problems. Finally we will conclude the section with MongoDB's limitations and the use cases where it's not a good fit.

# Use Cases

## Use Case 1 - Performance Monitoring

In this section we will explore how we can use MongoDB to store and retrieve performance data.

We'll focusing on

*What will be the data model that we will be using for storing the data. Retrieving will be simply reading from the respective collection. We will also be looking at how we can apply Sharding and replication for better performance and data safety.*



*We assume a monitoring tool which is collecting server wise defined parameter data in CSV format.*

*In general the monitoring tools either store the data as text files on a designated folder on the server or they can redirect their output to any reporting database server.*

*In this use case we are assuming that there's a scheduler which will be reading this shared folder path and importing the data within MongoDB database.*

### Schema Design

The first step in designing a solution is to decide on the schema. The schema depends on the format of the data which the monitoring tool is capturing.

A line from the log files may resemble as below

Node UUID	IP Address	Node Name	MIB	Time Stamp (ms)	Metric Value
-----------	------------	-----------	-----	-----------------	--------------

<b>3beb1a8b-040d-4b46-932a-2d31bd353186</b>	10.161.1.73	corp_xyz_sardar	IFU
---	-------------	-----------------	-----

1369221223384	0.2
---------------	-----

Simplest way is to store each line as a text as shown below

```
{  
  _id: ObjectId(...),  
  line: '10.161.1.73 - corp_xyz_sardar  
 [10/Oct/2000:13:55:36 -0700] "Interface Util" ...  
}
```

Although this captures the data but it is of no use as capturing data in this way makes no sense to the user for e.g. if we want to find out events from a particular server it'll require usage of regular expression which will further lead to full scan of the collection and hence is very inefficient.

Hence we will extract the data from the log file and store them as meaningful fields in MongoDB documents.

*We need to note that while designing the structure it's very important to use correct data type, this not only saves the space but will also have significant impact on the performance.* Say for example if we store the date and time field of the log as a string it'll not only use more bytes but also it'll be difficult to fire date range queries. Instead of using string if we store the date as UTC timestamp, it will take not only 8 bytes as opposed to 28 bytes of string also it'll be easier to execute date range queries. Hence using proper types for the data increases querying flexibility.

We will be using the following document for storing our monitoring data:

```
{  
  _id: ObjectId(...),  
  Host:,
```

```
Time:ISODate('),
ParameterName:'aa',
Value:10.23
}
```

 *The actual log data might have extra fields; if we capture all it'll lead to large documents which can lead to inefficient usage of the storage and memory hence while designing the schema we are omitting the details which are not required. It's very important to identify which fields need to be captured that will meet your requirement.*

In our scenario as mentioned in the sample document above the most important information that we are capturing which meets our reporting application requirement are:

1. Host
2. Timestamp
3. Parameter
4. Value

## **Operations**

Having designed the document structure next we will look at the various operations that we need to perform on the system.

## **Inserting Data**

The method used for inserting data depends on the way our application write concerns are.

1. If we are looking for fast insertion speed and can compromise on the data safety then the below command will be used  
> db.perfpoc.insert({Host:"Host1", GeneratedOn:  
new ISODate("2014-02-  
18T12:02Z"), ParameterName:"CPU", Value:13.13},  
w=0)

>

Though this command is the fastest option available as it doesn't wait for any acknowledgment of whether the operation was successful or not however in this we might have a risk of losing data.

2. If we want just an acknowledgment that at least the data is getting saved then we can issue the following command

```
>db.perfpoc.insert({Host:"Host1", GeneratedOn:  
new ISODate("2014-02-  
18T12:07Z"), ParameterName:"CPU", Value:13.23},  
w=1)  
>
```

Though this command acknowledges that the data is saved but it will not provide any safety against any adversity.

3. In our system the primary focus is to trade off increased insertion speed for data safety guarantees.

Hence we will be issuing the following command

```
> db.perfpoc.insert({Host:"Host1", GeneratedOn:  
new ISODate("2014-02-  
18T12:09Z"), ParameterName:"CPU", Value:30.01},j  
=true,w=2)  
>
```

In this we are not only ensuring that the data is getting replicated we are also enabling Journaling. Hence this will force the application to acknowledge that the data has replicated to two members of the replica set and in addition to this replication acknowledgment it also waits for a successful journal commit.

 *Though this is the safest option but it'll have severe impact on the insert performance, it is the slowest operation.*

## Bulk Insert

Inserting events in bulk is always beneficial when using stringent write concerns as in our case because this enables MongoDB to distribute the performance penalty incurred across a group of insert.

Hence if possible bulk inserts should be used for inserting the monitoring data as the data will be huge and will be getting generates in fraction of seconds hence grouping them together as a group and inserting will have better impact as in the same wait time multiple events will be getting saved.

Hence for this use case we will be grouping multiple events as using bulk insert. All write concern options apply to bulk inserts.

## Querying Performance data

We have seen how we can insert the event data. The value of maintaining data derives from being able to query the data to answer specific queries.

For e.g. we may want to view all the performance data associated with a specific field say a Host.

Hence in this part we will look at few query patterns for fetching the data and thereafter look at how we can optimize these operations.

*Query1: Fetch the performance data of a particular host*

```
> db.perfpoc.find({Host:"Host1"})
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" :
"Host1", "GeneratedOn"
: ISODate("2014-02-18T12:02:00Z"), "ParameterName" :
"CPU", "Value" : 13.13 }
{ "_id" : ObjectId("5302ffb609904770f56bd7ac"), "Host" :
"Host1", "GeneratedOn"
: ISODate("2014-02-18T12:07:00Z"), "ParameterName" :
"CPU", "Value" : 13.23 }
{ "_id" : ObjectId("5303003509904770f56bd7ad"), "Host" :
"Host1", "GeneratedOn"
: ISODate("2014-02-18T12:09:00Z"), "ParameterName" :
"CPU", "Value" : 30.01 }
>
```

*This can be used if the requirement is to analyze the performance of a host.*

*Here an index on the 'host' field will optimize performance.*

```
> db.perfpoc.ensureIndex({Host:1})
>
```

**Query2:** Fetch data within a date range say from 14<sup>th</sup> Feb 2014 to 20<sup>th</sup> Feb 2014

```
> db.perfpoc.find({GeneratedOn:{$gte: ISODate("2014-02-14"), "$lte": ISODate("2014-02-20")}})  
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" : "Host1", "GeneratedOn"  
.....  
>
```

*This is important if we want to consider and analyze the data collected for a specific date range.*

*Here an index on the ‘time’ field will optimize performance.*

```
> db.perfpoc.ensureIndex({GeneratedOn:1})  
>
```

**Query3:** Fetch data within a date range say from 14<sup>th</sup> Feb 2014 to 20<sup>th</sup> Feb 2014 for a specific host

```
> db.perfpoc.find({GeneratedOn:{$gte: ISODate("2014-02-14"), "$lte": ISODate("2014-02-20")}, Host: "Host1"})  
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" : "Host1", "GeneratedOn"  
.....  
>
```

*This is useful if we want to look at the performance data of a host for a specific time period.*

In such queries where we are having multiple fields involved the indexes that are used has a significant impact on the performance. Hence for the above query creating a compound index will be beneficial. We need to note that the fields order within the compound index has an impact.

*Let's understand the difference with an example.*

*Let's create a compound index as follows*

```
>
db.perfpoc.ensureIndex({"GeneratedOn":1,"Host":1})
>
```

*Next do an explain of this*

```
>           db.perfpoc.find({GeneratedOn:{$gte": ISODate("2014-02-14"), "$lte": ISODate("2014-02-20")}, Host: "Host1"}).explain()
{
  "cursor" : "BtreeCursor GeneratedOn_1_Host_1",
  "isMultiKey" : false,
  "n" : 3,
  "nscannedObjects" : 3,
  "nscanned" : 3,
  .....
}
>
```

*Drop the compound index*

```
> db.perfpoc.dropIndexes()
{
  "nIndexesWas" : 2,
  "msg" : "non-_id indexes dropped for collection",
  "ok" : 1
}
```

*Alternatively create the compound index with the fields reversed*

```
>
db.perfpoc.ensureIndex({"Host":1,"GeneratedOn":1})
>
```

*Do an explain*

```
>           db.perfpoc.find({GeneratedOn:{$gte": ISODate("2014-02-14"), "$lte": ISODate("2014-02-
```

```

20"}}, Host: "Host1").explain()
{
  "cursor" : "BtreeCursor Host_1_GeneratedOn_1",
  "isMultiKey" : false,
  "n" : 3,
  .....
}
>

```

*We can see the difference in the explain's output. Using explain() we can figure out the impact of indexes and accordingly decide on the indexes based on our application usage.* It's also recommended to have a single compound indexes covering maximum queries rather than having multiple single key indexes.

*Hence for our application usage and checking the explain statistics we will be using only one compound index on {'GeneratedOn':1, 'Host': 1} to cover all the above mentioned queries.*

#### **Query4:** Fetch count of performance data by Host and Day

*Listing of the data is good but mostly queries on performance data is to find out count or average or sum or other aggregate operations during analysis. Here we will see how we can use the aggregate command to select process and aggregate the results to satiate the need of powerful ad-hoc queries.*

*In order to explain this further lets write a query which will count the data per month.*

```

> db.perfpoc.aggregate(
... [
...   {$project:      {month_joined:      {$month:
"$GeneratedOn"}},},
...   {$group: {_id: {month_joined: "$month_joined"}, 
number: {$sum:1}}},
...   {$sort: {"_id.month_joined":1}}
... ]
... )

```

```
{
  "result": [
    {
      "_id": {
        "month_joined": 2
      },
      "number": 3
    }
  ],
  "ok": 1
}
>
```

*In order to optimize the performance we need to ensure that the filter field has an index. We have already created an index which covers the same, hence for this scenario we need not create any additional index.*

## **Sharding**

As the performance monitoring data set are humongous and sooner or later it will exceed the capacity of a single server. Hence we should consider using a shard cluster to take advantage of the MongoDB's automatic Sharding functionality.

In this section we will look at which shard key suits our use case of performance data properly so that the load is distributed across the cluster and no one server is overloaded.

The choice of shard key controls how MongoDB distributes data and the resulting systems capacity for writes and queries. The shard key ideally should have the following two characteristics:

1. *Insertions are balanced across the shard cluster*
2. *Most queries can be routed to subset of shards to be satisfied.*

Let's see which all fields can be used for Sharding. If we shard by

1. **Time field:** Choosing this though the data will be distributed evenly among the shards but neither the inserts nor the reads will be balanced.

*As in case of performance data the time field is in upward direction hence all the inserts will end up going to a single shard hence the write throughput will end up being same as in case of a standalone instance.*

*In case of reads also most reads will also end up on the same shard, assuming if we are interested in viewing the most recent data frequently.*

2. **Hashes:** We can also consider using a random value to cater to the above situations say we can consider hash of the `_id` field as the shard key.

*Though this will cater to the write situation of the above that is writes will be distributed but it will affect querying. In this case the queries must be broadcasted to all the shards and are not routable.*

3. An evenly distributed key in the data set say for e.g. **Host**.

*This has couple of advantages – writes will tend to be balanced among shards and reads will tend to be selective and local to a single shard if query selects on the host field.*

*However the biggest potential drawback is that all data collected for a single host must go to the same chunk since all the documents in it have the same shard key. This will not be a problem if the data is getting collected across all the hosts but if the monitoring collects a disproportionate amount of data for one host then we can end up with a large chunk which will be completely unsplittable causing an unbalanced load on one shard.*

4. Combining best of option 2 and 3.

We can have a compound shard key which is as follows `{host:1, ssk: 1}` where `host` is the host field of the document and `SSK` is hash of `_id` field.

*Using this shard key data is largely distributed by the host which makes most queries that access the host field local to a single shard or group of shards. At the same time, if there is not sufficient distribution for specific values of path, the ssk makes it possible for MongoDB to create chunks that distribute data across the cluster.*

In most situations, these kinds of keys provide the ideal balance between distributing writes across the cluster and ensuring that most queries will only need to access a select number of shards.

Hence we are using the fourth option.

*Nevertheless, the best way to select a shard key is to analyze the actual insertions and queries from your own application.*

## ***Managing the data***

As the performance data is humongous and it continues to grow we define a data retention policy which states that we will be maintaining the data for the specified period say we will have data for 6 months.

Hence the question which arises next is how we will remove the old data.

We can follow the following patterns

1. **Use Capped collection.** Though capped collections can be used to store the performance data but in the current version (2.4 at the time of writing) it is not possible to shard capped collection.
2. **Use TTL collections.** This pattern creates a collection similar to capped collection which can be sharded.

In this case a Time to live index is defined on the collection which enables MongoDB to periodically remove() old documents from the collection. However this do not possess the performance advantage of the capped collection and in addition the remove() may lead to data fragmentation.

3. **Multiple collections to store the data.** Third pattern is to have Day wise collection created which will have documents storing that day's performance data. Hence this ways we will end up having multiple collections within a database. Though this will complicate the querying as in order to fetch two day's data we might need to read from two collections but dropping a collection is fast and space can be reused effectively without any data fragmentation. In our use case we are using this pattern for managing the data.

# Use Case 2 – Social Networking

In this section we will explore how we can use MongoDB to store and retrieve data of a social networking site.



*This use case is basically a friendly social networking sites which allows users to share their statuses and photos hence the solution provided here for this use case assumes the following*

1. *User can choose whether or not to follow another user*
2. *User can decide on the circle of users with whom he wants to share updates, we have the following circles Friends, Friends of Friends and Public.*
3. *The updates allowed are status updates and photos.*
4. *A user profile displays interests, gender, age and relationship status*

## Schema Design

The solution which we are providing is aimed to minimize the number of documents that must be loaded in order to display any given page.

The application in question has two main pages first page which displays the user's wall which is intended to display posts created by or directed to a particular user and the other page is the social news page where all the notifications and activities of all the people who are following the user or whom the user is following will be displayed.

In addition to these two pages we have a user profile page which displays the users profile related details with information of his friend group who are following him or whom he is following. In order to cater to this requirement the schema for this use case consists of the following collections.

1<sup>st</sup> collection is “*user.profile*” which stores the user’s profile related data

```
{  
  _id: “User Defined unique identifier”,  
  UserName: “user name”  
  ProfilDetaile:  
    {Age:..., Place:..., Interests: ...etc},  
  FollowerDetails: {  
    “User_ID”: {name:..., circles: [circle1, circle2]},  
    ....  
  },  
  CirclesUserList: {  
    “Circle 1”:  
      {“User_Id”: {name: “username”}, .....  
      }, .....  
    } ,  
  ListBlockedUserIDs: [“user1”,...]  
}
```

In this case we are manually specifying the `_id` field. The follower details are list of users following the user. Circles User List consists of circle wise list of users the user is following. Blocked consist of users whom the user has blocked from viewing his/her update.

2<sup>nd</sup> collection is “*user.posts*” collection with the following schema.

```
{  
  _id: ObjectId(...),  
  Postedby: {id: “user id”, name: “user name”},  
  VisibleTocircles: [],  
  PostType: “post type”,  
  ts: ISODate(),  
  Postdetail: {text: “”,  
  Comments_Doc:  
    [  
      {Commentedby: {id: “user_id”, name: “user  
name”}, ts: ISODate(), Commenttext: “comment  
text”} , .....,  
    ]  
  }  
}
```

*This collection basically is for displaying all users activities. “by” provides the information of the user who posted the post. “Circles” controls the visibility of the post to other users. “Type” is used to identify the content of the post. “ts” is the datetime when post was created. “detail” contains the post text as well as it has all comments embedded within it. A “comment” document consist of the following details: “by” provides details of the user id and name who commented on the post, “ts” is the time of comment and “text” is the actual comment posted by the user.*

3<sup>rd</sup> collection is “*user.wall*” which is used for rendering the user’s wall page in fraction of seconds. This collection fetches data from the 2<sup>nd</sup> collection and stores it in a summarized format for fast rendering of the wall page.

The collection has the following format

```
{  
  _id: ObjectId(...),  
  User_id: "user id"  
  PostMonth: "201402",  
  PostDetails: [  
    {  
      _id: ObjectId(..), ts: ISODate(), by: {_id: ...,  
      Name:... }, circles: [...], type: ....  
      , detail: {text: "..."}, comments_shown: 3  
      , comments: [  
        {by: {_id:.., Name:....}, ts: ISODate(), text:""},  
        .....]  
      }, ....] } }
```

As we can see we are maintaining this document per user per month. Number of comments which will be visible on the first time is limited (for this example it’s 3) if more comments need to be displayed for that particular post then the 2<sup>nd</sup> collection need to be queried.

Hence it’s kind of a summarized view for quick loading of the users wall page.

The 4<sup>th</sup> collection is “*social.posts*” which is used for quick rendering of the social news screens. This is the screen where all posts are getting displayed.

*Like the 3rd collection, 4th collection is also a dependent collection.*

This schema includes much of the same information as the *user.wall* information, so this document has been abbreviated for clarity:

```
{  
  _id: ObjectId(...),  
  user_id: "user id",  
  postmonth: '2014_02',  
  postlists: [ ... ]  
}
```

## ***Operations***

These schemas optimize for read performance.

## **Viewing Posts**

Since *social.posts* and *user.wall* collections are optimized for rendering the news feed or wall posts in fraction of seconds the query is fairly straightforward.

Also both the collections have similar schema hence the fetch operation can be supported by the same code. Find below a pseudo code for the same. The function takes as parameter the following

1. the collection that need to be queried
2. the user whose data need to be viewed and
3. Month is an optional parameter if specified it should list all the posts of the date <= the month specified.

```

Function Fetch_Post_Details
(Parameters: CollectionName, View_User_ID,
Month)
SET QueryDocument to {"User_id": View_User_ID}
IF Month IS NOT NULL
APPEND Month Filter [{"Month":{$lte:Month}}] to QueryDocument
Set O_Cursor = (resultset of the collection after applying the QueryDocument filter)
Set Cur = (sort O_Cursor by "month" in reverse order)
while records are present in Cur
Print record
End while
End Function

```

The above function retrieves all the posts on the given user's wall or news feed in reverse-chronological order.

While rendering posts there are certain checks which we need to apply. Following are list of few.

First when the user is viewing his or her page then while rendering the post on the wall we need to check whether the same can be displayed on their own wall. A user wall contains post which he has posted or the posts of the users they are following. The following function takes two parameters

1. The user who's wall is it and
2. The post which is being rendered.

```
function Check_VisibleOnOwnWall
(Parameters: user, post)
While Loop_User IN user.Circles List
    If post by = Loop_User
        return true
    else
        return false
    end while
end function
```

The function above loop through the circles list specified in user.profile collection and if the mentioned post is posted by the user in the list it returns true.

In addition we also need to take care of the users in the blocked list of the user:

```
function ReturnBlockedOrNot(user, post)
    if post by user id not in user blocked list
        return true
    else
        return false
    endfunction
```

We also need to take care of the permission checks when the user is viewing another user's wall.

```

Function visibleposts(parameter user, post)
  if post circles is public
    return true
  If post circles is public to all followed users
    Return true
  set listofcircles = followers circle whose user_id
  is the post's by id.

  if listofcircles in post's circles
    return true
  return false

end function

```

This function first check whether the post's circle is public, if it's public then the post will be displayed to all user. If the post circles are set to be public to all followers then it will be displayed to the user if he/she is following the user. If neither is true then it gets the circle of all the users who are following the logged in user. If the list of circle is in posts circle list this implies the user is in a circle receiving the post and the post will be visible. If neither condition is met the post will not be visible to the user.

In order to have better performance we'll need an index on user\_id, month in both the social.posts and user.wall collections.

## Create Comment

To create a comment by user on a given post containing the given text, we'll need to execute code similar to the following:

*Function postcomment(*

*Parameters: commentedby, commentedonpostid,  
commenttext)*

*Set commentedon to current datetime*

*Set month to month of commentedon*

*Set comment document as {"by": {id:  
commentedby[id],  
commentedby["name"]}, "ts": commentedon,  
"text": commenttext}*

*Update user.posts collection. Push comment  
document.*

*Update user.walls collection. Push the comment  
document.*

*Increment the comments\_shown in user.walls  
collection by 1.*

*Update social.posts collection. Push the  
comment document.*

*Increment the comments\_shown counter in  
social.posts collection by 1.*

*End function*

Since we are displaying only maximum three comments in both the dependent collection i.e. user.wall and social.posts collection. We need to run the following update statement periodically

```

Function MaintainComments
  SET MaximumComments = 3
  Loop through social.posts
    If           posts.comments_shown      >
MaximumComments
      Pop the comment which was inserted first
      Decrement comments_shown by 1
    End if
  Loop through user.wall
    If           posts.comments_shown      >
MaximumComments
      Pop the comment which was inserted first
      Decrement comments_shown by 1
    End if

  End loop
End Function

```

To quickly execute these updates, we need to create indexes on posts.id and posts.comments\_shown.

## Create new

The basic sequence of operations in this code is as follows:

1. The post is first saved into the “system of record,” the user.posts collection.
2. Next the user.wall collection is updated with the post.
3. Finally the social.posts collection of everyone who is circled in the post is updated with the post.

```

Function createnewpost
(parameter createdby, posttype, postdetail,
circles)
Set ts = current timestamp.
Set month = month of ts
Set post_document = {“ts”: ts, “by”:
{id:createdby[id], name: createdby[name]},
“circles”:circles, “type”:posttype,
“details”:postdetails}
Insert post_document into users.post collection
Append post_document into user.walls
collection
Set userlist = all users who’s circled in the post
based on the posts circle and the posted user id
While users in userlist
Append post_document to users social.posts
collection
End while
End function

```

## **Sharding**

Scaling can be achieved by Sharding all the four collection mentioned above.

Since the user.profile, user.wall, social.posts contains user specific documents hence user\_id is perfect shard key for these collections.

\_id is the best shard key for the users.post collection.

# Limitations and Not Applicable Ranges

In this section we will briefly highlight the limitations and list down use cases which are not so good fit for MongoDB.

## Limitations

### ***MongoDB space is too large***

The first limitation which is highlighted almost at all the forums and which we need to be aware of is the issue of disk space.

MongoDB space is too large i.e. the files in the data directory are larger than the actual data in the database.

The following are the probable causes for the same

#### **Preallocated data files**

This is by design in order to prevent file system fragmentation.

Also Pre-allocating data files in the background prevents significant delays when a new database file is next allocated.

*MongoDB names the first data file <dbname>.0, the next <dbname>.1, etc. The first file mongod allocates is 64 megabytes, the next 128 megabytes, and so on, up to 2 gigabytes, at which point all subsequent files are 2 gigabytes.*

The data files include files with allocated space but that hold no data. mongod may allocate a 1 gigabyte data file that may be 90% empty.

Mostly for larger databases, unused allocated space is small compared to the database.

*This option can be disabled with the noprealloc run time option. However noprealloc is not intended for use in production environments: only use noprealloc for testing and with small data sets where you frequently drop databases.*

**Oplod** - If this mongod is a member of a replica set, the data directory includes the oplog.rs file, which is a preallocated capped collection in the local database.

*The default allocation is approximately 5% of disk space on 64-bit installations.*

**Journal** – The data directory contains the journal files which stores the write operations on disk before the same can be applied to the databases by MongoDB.

*MongoDB pre-allocates 3GB of data for journaling which is over and above the actual database size(s) making it not-fit for small installations. The workaround available for this is to use –smallflags in your command line flags or /etc/mongod.conf files until running in an environment where we have the required disk space. But this feature makes it not-fit for small installations.*

**Empty Records** – when the documents or collections are deleted, the space is never returned back to the operating system instead MongoDB maintains a list of these empty records and can be reused.

*To reclaim this deleted space either compact or repairDatabase option can be used but we need to be aware of the fact that both this options require additional disk space to run.*

## **Memory Issues**

MongoDB manages memory by memory mapping your entire data set, leaving page cache management and faulting up to the kernel.

The result is that memory usage can't be effectively reasoned about, and performance is non-optimal.

1. *Indexes are Memory heavy i.e. Indexes takes up lot of RAM. Since this are B-Tree indexes defining lot many indexes can lead to faster consumption of system resources.*

2. *A consequence of using virtual memory mapping is that memory for the data will be allocated automatically, as needed. This makes it trickier to run the database in a shared environment. As with*

*database servers in general, MongoDB is best run on a dedicated server.*

## **32-bit vs. 64-bit**

MongoDB ships with two versions – 32-bit and 64-bit.

Since MongoDB uses memory mapped files hence the 32-bit versions can only store around 2GB of data.

If needed more data to be stored then use 64-bit build.

Hence either we should use 64-bit or understand this size limitation of 32-bit before proceeding.

## **BSON Document**

### **Size limits**

Like most other databases, there are limits to what you can store in a document.

*The current versions (2.4 at the time of writing) support documents up to 16 megabytes in size.*

10gen's opinion on this is that if you are hitting this limit then either your schema design is wrong or you should be using GridFS, which allows arbitrarily sized documents.

The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth.

### **Nested Depth Limit**

MongoDB supports no more than 100 levels of nesting for BSON documents.

### **Uncompressed field names**

If you store 1,000 documents with the key “foo”, then “foo” is stored 1,000 times in your data set. Although MongoDB supports any arbitrary document, in practice most of your field names are similar.

*It is considered good practice to shorten field names for space optimization.*

# **Namespaces Limits**

## **Namespace Length**

Each namespace, including database and collection name, must be shorter than 123 bytes.

## **Number of Namespaces**

The limitation on the number of namespaces is the size of the namespace file divided by 628. A 16 megabyte namespace file can support approximately 24,000 namespaces. Each index also counts as a namespace.

## **Size of Namespace File**

Namespace files can be no larger than 2047 megabytes.

*By default namespace files are 16 megabytes. You can configure the size using the nssize option.*

# **Indexes Limit**

## **Index Size**

Indexed items can be no larger than 1024 bytes.

## **Number of Indexes per Collection**

A single collection can have no more than 64 indexes.

## **Index Name Length**

The names of indexes, including their namespace (i.e database and collection name) cannot be longer than 128 characters.

*The default index name is the concatenation of the field names and index directions.*

*You can explicitly specify an index name to the ensureIndex() helper if the default index name is too long.*

## **Unique Indexes in Sharded Collections**

MongoDB does not support unique indexes across shards, except when the unique index contains the full shard key as a prefix of the index.

In these situations MongoDB will enforce uniqueness across the full key, not a single field.

### **Number of Indexed Fields in a Compound Index**

There can be no more than 31 fields in a compound index.

### ***Capped Collections Limit***

#### **Maximum Number of Documents in a Capped Collection**

If you specify a maximum number of documents for a capped collection using the max parameter to create, the limit must be less than 2<sup>32</sup> documents. If you do not specify a maximum number of documents when creating a capped collection, there is no limit on the number of documents.

# **Sharding Limitations**

## **Shard early to avoid any issues**

Sharding is the way of splitting data across machines. To shard MongoDB splits the chosen collection into chunks based on the shard key and distributes them amongst the shards automatically.

However if sharding is implemented late it can cause slowdowns of the servers as the splitting and migration of chunks takes time and resource.

Hence the simple solution is use a tool such as MMS to keep an eye on MongoDB, make a best guess of your capacity (flush time, queue lengths, lock percentages and faults are good gauges) and shard before you get to 80% of your estimated capacity.

## **Shard Key cannot be updated**

Shard key cannot be updated once the document is inserted in the collection as MongoDB uses shard keys to work out which shard a particular document should be on. Hence if we want to change the shard key of a document the suggested solution is to remove the document and reinsert it.

## **Shard Collection limit**

Shard collections before it reaches 256GB as MongoDB won't allow to shard a collection that has grown bigger than 256GB.

## **Select the correct shard key**

MongoDB requires you to choose a key to shard your data on. Choosing a correct shard key is very important because once the shard key is chosen it's not easy to correct.

 *What is the wrong shard key depends on your application - but a common example would be using a timestamp for a news feed. This causes one shard to end up 'hot' by having data constantly inserted into it, migrated off and queried too. The common process for altering the shard key is: dump and restore the collection.*

## **Security Limitations**

### **No authentication by default**

MongoDB doesn't have authentication by default. It's expected that mongod is running in a trusted network and behind a firewall.

*However authentication is fully supported and can be enabled easily.*

### **Traffic to and from MongoDB is not encrypted**

The connections to and from MongoDB are not encrypted by default hence when running in a public network consider encrypting the communication otherwise it can pose threat to your data.

*Though the default distribution of MongoDB does not contain support for SSL. To use SSL, you must either build MongoDB locally passing the “--ssl” option to scons or use MongoDB Enterprise.*

## **Write and Read Limitations**

### **Safe off by default**

MongoDB allows very fast writes and updates by default. The tradeoff is that you are not explicitly notified of failures.

*If you want to know if something succeeded, you have to manually check for errors using getLastErrors.*

For cases where you want an error thrown if something goes wrong, it's simple in most drivers to enable "safe" queries which are synchronous.

*If you need more performance than a ‘fully safe’ synchronous write, but still want some level of safety, you can ask MongoDB to wait until a journal commit has happened using getLastErrors with ‘j’. The journal is flushed to disk every 100 milliseconds, rather than 60 seconds as with the main store.*

Hence we have to use safe writes or use getLastErrors if you want to confirm writes

### **Case sensitive queries**

MongoDB by default is case sensitive hence querying using strings may not quite work as expected.



*For example, db.people.find({name: 'Russell'}) is different to db.people.find({name: 'russell'}). The solution is to make sure your data is in a known case - which is the ideal solution. You can also use regex searches like db.people.find({name: /russell/i}) although these aren't ideal as they are relatively slow.*

## Type sensitive fields

When you try and insert data with an incorrect data type into a traditional database, it will generally either error or cast the data to a predefined value. However with MongoDB there is no enforced schema for documents, so MongoDB can't know you are making a mistake. If you write a string, MongoDB stores it as a string. If you write an integer, it stores it as an integer.

*Hence make sure you use the correct type for your data*

## No JOIN

MongoDB does not support joins; If you need to retrieve data from more than one collection you must do more than one query.

*If you find yourself doing too many queries, you can generally redesign your schema to reduce the overall number you are doing. Documents in MongoDB can take any format, so you can denormalize your data easily. Keeping it consistent is down to your application however.*

## Transactions

MongoDB only supports single document atomicity which means it can be hard without being creative to model things which have shared state across multiple collections.

There is no built in support for transactions over multiple documents

## **Replica set Limitations**

### **Using an even number of Replica Set members**

Replica Sets are an easy way to add redundancy and read performance to your MongoDB cluster. Data is replicated between all the nodes and one is elected as the primary. If the primary fails, the

other nodes will vote between themselves and one will be elected the new primary.

It can be tempting to run with two machines in a replica set; it's cheaper than three and is pretty standard way of doing things with RDBMSs.

However due to the way voting works with MongoDB, you must use an odd number of replica set members.

*If you use an even number and one node fails the rest of the set will go read-only. This happens as the remaining machines will not have enough votes to get to a quorum. If you want to save some money, but still support failover and increased redundancy, you can use arbiters.*

Arbiters are a special type of replica set member - they do not store any user data (which means they can be on very small servers) but otherwise vote as normal. Hence only use an odd number of replica set members and be aware that arbiters can reduce the costs of running a redundant setup

*Running a replica set is a great way to make your system more reliable and easier to maintain. An understanding of what happens during a node failure or failover is important.*

*Replica Sets work by transferring the oplog - a list of things that change in your database (updates, inserts, removes, etc.) - and then replaying it on other members in the set. If your primary fails and later comes back online, it will roll back to the last common point in the oplog. At any point during this process, newer data that may have existed will be removed from the database and placed in a special folder in your data directory called 'rollback' for you to manually restore. If you don't know about this feature, you may find that data goes missing. Each time you have a failover you should check this folder. Manually restoring the data is really easy with the standard tools that ship with MongoDB. Hence remember 'missing data' after a failover will be in the rollback directory*

## MongoDB Not applicable Range

MongoDB is not suitable for the following

1. Highly transactional system such as banks or accounting systems. Traditional relational database is still more suitable for applications that need a large number of atomic complex matters.
2. Traditional business intelligence applications - issue-specific BI database would generate highly optimized query. For such applications, the data warehouse may be a more appropriate choice.
3. Applications requiring complex SQL queries
4. MongoDB does not support transactional operations, affairs demanding system (banking system) certainly cannot use it.

# Summary

In this chapter we have used two use cases to look at how MongoDB can be used to solve such problems. Finally we concluded the chapter by listing down MongoDB's limitations and the use cases where it's not a good fit.

In the next chapter we will cover How To's of MongoDB.

## MongoDB How To's

*"Getting started with MongoDB is easy but once we start developing application we will come across various issues and will get stuck. This chapter enables you as a user to handle the failure, be aware of the How To's which will help resolving issues ranging from design, implementation to data safety and monitoring."*

In the previous chapters we have become acquainted with MongoDB.

Getting started with any new technology is easy and so is the case with MongoDB but the complexity arises once we start developing, once we start getting more inside it we will come across various issues.

Hence the intent of this chapter is to not only enable users in handling well known issues using other user's experiences but also to provide various How To's which can help the users journey with MongoDB a smooth ride.

# How To's

We are aware of MongoDB by now, we know that it works with documents, uses RAM for storing the data to enhance the performance, uses Replication and Sharding further for providing data safety and scalability. We should know what suits best for us.

This part of the chapter will cover tips which the user should be aware of starting from the deployment strategy to enhancing querying to data safety and consistency to monitoring.

## Deployment

While deciding on the deployment strategy, keep the following tips in mind so that the hardware sizing is done appropriately, the following also enables you to decide on whether to use Sharding and replication.

**#1 :** The most important will be your current and anticipated data set size.

This will be the primary driver for your choice of individual physical node resource needs as well as a guide for your Sharding plans.

**#2 :** The next most important thing to consider is the importance of your data and how tolerant you will be of the possibility of lost or lagging data (especially in replicated scenarios).

**#3 :** Memory sizing i.e. identifying memory needs and accordingly take care of the RAM.

*MongoDB (like many data oriented applications) works best when the data set can reside in memory.*

Nothing performs better than a MongoDB instance that does not require disk I/O.

*Whenever possible select a platform that has more available RAM than your working data set size. If your data set exceeds the*

*available RAM for a single node, then consider using Sharding to increase the amount of available RAM in a cluster to accommodate the larger data set. This will maximize the overall performance of your deployment.*

Page faults can indicate that you may be exceeding available RAM in your deployment and you may need to increase your available RAM.

**#4 : Disk Type:** If speed is not your primary concern, or if you have a data set that is far larger than any available in-memory strategy can support, then selecting the proper disk type for your deployment is important. IOPS will be the key in selecting your disk type and obviously the higher the IOPS the better the performance of MongoDB.

*Local disks should be used if possible as network storage can cause high latency and poor performance for your deployment. Whenever possible it is also advised that you utilize RAID 10 when creating disk arrays.*

**#5 : CPU -** Clock speed and the amount of available processors becomes a consideration if you anticipate using map reduce.

*However, it has been noted that when running a MongoDB instance with the majority of the data being in memory, clock speed can have a major impact on overall performance. If you are running under any of these circumstances and would like to maximize your operations per second, consider a deployment strategy that includes a CPU with a high clock/bus speed.*

**#6 :** If high Availability is one of the requirements then we need to use Replication. Replication provides high availability of your data if node fails in your cluster. It should be standard to replicate with at least 3 nodes in any MongoDB deployment.

*The most common configuration for replication with 3 nodes is a 2x1 deployment having 2 nodes in a single data center with a backup server in a secondary data center.*

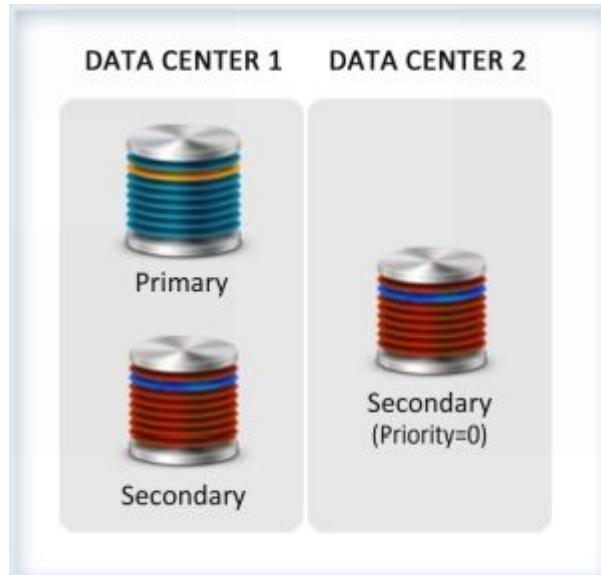


Fig 10-1: MongoDB 2\*1 deployment

## ***Hardware Suggestions from MongoDB 10gen site***

The following suggestions are only intended to provide high-level guidance for hardware for a MongoDB deployment.

The specific configuration of your hardware will be dependent on your data, your queries, your performance SLA, your availability requirements, and the capabilities of the underlying hardware components.



*Note: MongoDB was specifically designed with commodity hardware in mind and has few hardware requirements or limitations. Generally speaking, MongoDB will take advantage of more RAM and faster CPU clock speeds.*

**MEMORY** – Since MongoDB makes extensive use of RAM to increase performance. Hence as a general rule of thumb, the more RAM, the better.

As workloads begin to access data that is not in RAM, the performance of MongoDB will degrade. MongoDB will use as much RAM as possible until it exhausts what is available.

**STORAGE** - MongoDB does not require shared storage i.e. storage area networks etc. MongoDB can use local attached storage as well as solid state drives (SSDs).

Most disk access patterns in MongoDB do not have sequential properties, and as a result, customers may experience substantial performance gains by using SSDs. Good results have been tested and strong price to performance with SATA SSD and with PCI.

Commodity SATA spinning drives are comparable to higher cost spinning drives due to the non-sequential access patterns of MongoDB: rather than spending more on expensive spinning drives, that money may be more effectively spent on more RAM or SSDs.

Another benefit of using SSDs is that they provide a more gentle degradation of performance if the working set no longer fits in memory.

Most MongoDB deployments should use RAID-10.

RAID-5 and RAID-6 do not provide sufficient performance.

RAID-0 provides good write performance, but limited read performance and insufficient fault tolerance.

MongoDB's replica sets allow deployments to provide stronger availability for data, and should be considered with RAID and other factors to meet the desired availability SLA.

**CPU** - MongoDB performance is typically not CPU-bound. As MongoDB rarely encounters workloads able to leverage large numbers of cores, it is preferable to have servers with faster clock speeds than numerous cores with slower clock speeds.

## ***Few points to be noted***

In order to summarize this section, while choosing hardware for MongoDB, consider the following important points:

1. More RAM and a faster CPU clock speed are important to productivity.
2. As it doesn't perform high amounts of computation, increasing the number cores helps but does not provide a high level of marginal return.
3. It has good results and good price/performance with SATA SSD (Solid State Disk) and with PCI (Peripheral Component

Interconnect).

4. Commodity (SATA) spinning drives are often a good option as the speed increase for random I/O for more expensive drives is not that dramatic (only on the order of 2x), spending that money on SSDs or RAM may be more effective.
5. MongoDB on NUMA Hardware – This point only applies to Linux, and does not apply to deployments where mongod instances run other UNIX-like systems or on Windows.

MongoDB and NUMA, Non-Uniform Access Memory, do not work well together. When running MongoDB on NUMA hardware, disable NUMA for MongoDB and run with an interleave memory policy. NUMA can cause a number of operational problems with MongoDB, including slow performance for periods of time or high system processor usage.

# Coding

Once the hardware is thought of next we need to take care of the tips while coding with the database.

**#7 :** The first point is to think of the data model to be used for the given application requirement i.e. decide on embedding or referencing or mix of both. For more on this please look at chapter on Data Model. There's a tradeoff between fast performance and guaranteed immediate consistency. Hence decide based on your application.

**#8 :** Avoid application pattern that lead to unbound growth of document Size. The maximum BSON document size in MongoDB is 16MB. User should avoid application patterns that allow documents to grow unbounded. For instance, applications should not typically update documents in a way that causes them to grow significantly after they have been created, as this can lead to inefficient use of storage.

If the document size exceeds its allocated space, MongoDB will relocate the document on disk. This automatic process can be resource intensive and time consuming, and can unnecessarily slow down other operations in the database.



For example, in a blogging application it would be difficult to estimate how many responses a blog post might receive from readers. Furthermore, it is typically the case that only a subset of comments is displayed to a user, such as the most recent or the first 10 comments. Rather than modeling the post and user responses as a single document it would be better to model each response or groups of responses as a separate document with a reference to the blog post.

Another example is product reviews on an e-commerce site. The product reviews should be modeled as individual documents that reference the product. This approach would also allow the reviews to

*reference multiple versions of the product such as different sizes or colors.*

**#9** Design documents in a way it will be needed in the future. Though MongoDB provides option of appending new fields within the documents as and when required however it has a drawback as when the new fields are introduced there might be a scenario where the document might not fit in the current space available leading to MongoDB finding a new space for the document and moving it there which might take time. Hence it is always efficient to create all the fields at the start if you are aware of the structure irrespective of the fact whether you have data available at that time or not. As highlighted above reason being space will be allotted to the document and whenever value is there it only need to be updated, doing so MongoDB will not have to look for space for them, it merely updates the values entered which is much faster.

**#10** Create document with the anticipated size wherever applicable. This point is also to ensure that enough space is allotted to the document and any further growth doesn't lead to hopping here and there for space.

This can be achieved by using a garbage field which contains string of the anticipated size while initially insert the document and then immediately unset that field:

```
> mydbc.col.insert({"_id" : ObjectId(..), .....,  
"tempField" : stringOfAnticipatedSize})  
> mydbc.col.update({"_id" : ...}, {"$unset" :  
{"tempField" : 1}})
```

**#11** Arrays vs. Subdocuments - Subdocuments should always be used in a scenario when you know and will always know the name of the fields that you are accessing otherwise we should consider using Arrays.

**#12** If we want to query for information that must be computed and is not explicitly present in the document then the best choice is to make the information explicit in the document. As MongoDB is designed to just store and retrieve the data, it does no computation and any

trivial computation is pushed to the client leading to performance issues.

**#13** Avoid \$Where as much as possible because it's an extremely time and resource intensive operation.

**#14** Use the Correct data Types while designing the documents for e.g. a number should be stored as a Number data type only and not as an String data type. Because if we use string it will not only take more space to store data but it will also have an impact on the operations that can be performed on the data.

### # 15 MongoDB Strings are Case sensitive

MongoDB strings are case sensitive. Hence a search for "joe" will not find "Joe".

Hence when doing a string search, consider using the following:

1. storing data in a normalized case format, or
2. using regular expressions ending with /I while searching
3. and/or using \$toLower or \$toUpper in the aggregation framework

### # 16 Deciding on the \_id field

If you are using your own unique key as an \_id then though this will save a bit on the space and will be useful if you were planning to index on the key but you need to keep the following things in mind when deciding to use your own key as \_id

1. You must ensure the uniqueness of the key
2. Also consider how the insertion order will be for your key as the insertion order will identify how much RAM will be used for maintaining this index.

### # 17 Retrieve the fields as and when needed.

Retrieving whole documents is the equivalent of “select \* from table” in SQL and it is just as bad for performance. When you are fulfilling hundreds or thousands of requests per second, it is certainly advantageous to only retrieve the fields you need.

**# 18** Use GridFS only for storing data which is larger than what can fit in a single document or is big to load at once on the client such as Videos. Anything that will be streamed to a client is a good candidate for GridFS.

### # 19 Use TTL to delete documents

If documents in a collection should only persist for a pre-defined period of time, the TTL feature can be used to automatically delete

documents of a certain age rather than scheduling a process to check the age of all documents and run a series of deletes.



*For example, if user sessions should only exist for one hour, the TTL can be set to 3600 seconds for a date field called **lastActivity** that exists in documents used to track user sessions and their last interaction with the system. A background thread will automatically check all these documents and delete those that have been idle for more than 3600 seconds. Another example for TTL is a price quote that should automatically expire after a period of time.*

**# 2 0** Use Capped Collections if require high throughput based on insertion orders. In some scenarios a rolling window of data should be maintained in the system based on data size. *For example, store log information from a high-volume system in a capped collection to quickly retrieve the most recent log entries without designing for storage management*

**# 2 1** Handle Data inconsistency - MongoDB's flexible schema can lead to inconsistent data if not taken care. *For example the ability to duplicate data (embedded documents) if not updated properly can lead to data inconsistency and so on.* Hence it's very important to check for data consistency.

**# 2 2** Exception Handling - Though MongoDB handles seamless failover then also as per good coding practice the application should be well written to handle any exception and gracefully handle such situation .

## Application Response Time optimization

Now once we have started developing the application one of the most important requirements is to have a small response time i.e. the application should respond instantly to anything we do.

Hence we can use the following tips for optimization purposes.

**# 2 3** Avoid disk access and page faults as much as possible - Proactively figure out the data set size the application will be expected to deal with and add more RAM in order to avoid page faults and disk read. Also program an application in such a way that it mostly access data available in memory and page faults happens infrequently.

**# 2 4** Create index on the queried fields - if we create an index on the filter which we are executing then the way the index is stored in memory will lead to less consumption of RAM and hence will have a positive effect on the queries.

**#25** Create Covering indexes if the application involves queries which return few fields as compared to the complete document structure.

**#26** Having one compound index that can be used by maximum queries will also save on the RAM as instead of loading multiple indexes in the RAM one index will only suffice.

**#27** Use Trailing Wildcards in Regular expressions to reap in benefits from the associated Index. Trailing wildcards work well, but leading wildcards do not because the indexes are ordered.

**# 28:** Try and create low selectivity indexes. An index should radically reduce the set of possible documents to select from. *For example, an index on a field that indicates male/female is not as beneficial as an index on zip code, or even better, phone number.*

**# 29** Indexing is not always good.

Like most database management systems, indexes are a crucial mechanism for optimizing system performance in MongoDB.

And while indexes will improve the performance of some operations by one or more orders of magnitude, they have associated costs in the form of slower updates, disk usage, and memory usage.

Users should always create indexes to support queries, but should take care not to maintain indexes that the queries do not use. Each index incurs some cost for every insert and update operation: if the application does not use these indexes, then it can adversely affect the overall capacity of the database.

This is particularly important for deployments that have insert-heavy workloads.

**# 30** Documents should be designed in a hierarchical fashion where related things are grouped together and are depicted as hierarchy wherever applicable as it will enable MongoDB to find the desired information without scanning the entire document.

**# 31** When applying an AND operator we should always query from small resultset to a larger resultset as this will lead to querying small number of documents. Hence if you are aware of the most restrictive condition then that condition should go first.

**# 32** whereas when querying with OR we should move from Larger resultset to smaller resultset as this will limit the search space for subsequent queries

**# 33** Working set should fit in RAM

# Data Safety

We have covered what we need to keep in mind while deciding on our deployment; we also looked at few important tips which the user needs to keep in mind for good performance. Next we will see tips for data safety and consistency.

**#34:** Replication and Journaling are two approaches that are provided for data safety. *Generally it's recommended to run the production setup using replication rather than running it using a single server. And have at least one of the servers journaled. When it is not possible to have replication enabled and you are running on a single server then in that scenario Journaling gives you data safety.* The MongoDB Explained chapter explains how write works with Journaling enabled.

**#35** Repair should be the last resort for recovering data in case of a server crash. Though the database might not be corrupted after running repair but it will not be containing all the data.

**#36** Use Safe Writes by using getLastError command.

**#37** For a safe write in a replicated environment set *W* to majority to ensure that the writes is replicated to majority of the set. Though this will slow down the write operations but the write will be safe.

**#38** Always specify **wtimeout** along with **w** when issuing the command in order to *avoid the infinite waiting time*.

**#39** Always run MongoDB in a trusted environment, with network rules that prevent access from all unknown machines, systems, or networks.

# Administration

**#40:** Take instant-in-time backups of durable servers

To take a backup of a database with journaling enabled, you can take a file system snapshot or do a normal fsync+lock and then dump.

Note that you can't just copy all of the files without fsync and locking, as copying is not an instantaneous operation.

**#41** Repair should be used to Compact Databases as it basically does a mongodump and then a mongorestore, making a clean copy of your data and, in the process, removing any empty “holes” in your data files.

## # 4 2 Profiling

MongoDB provides a profiling capability called Database Profiler, which logs fine-grained information about database operations.

The profiler can be enabled to log information for all events or only those events whose duration exceeds a configurable threshold (whose default is 100ms).

*Profiling data is stored in a capped collection where it can easily be searched for interesting events – it may be easier to query this collection than parsing the log files.*

## # 4 3 Use Explain Plan

Queries are automatically optimized by MongoDB to make evaluation of the query as efficient as possible.

Evaluation normally includes the selection of data based on predicates, and the sorting of data based on the sort criteria provided.

Generally MongoDB makes use of one index in resolving a query. The query optimizer selects the best index to use by periodically running alternate query plans and selecting the index with the lowest scan count for each query type. The results of this empirical test are stored as a cached query plan and periodically updated.

MongoDB provides an explain plan capability that shows information about how a query was resolved, including:

- The number of documents returned.
- Which index was used?
- Whether the query was covered, meaning no documents needed to be read to return results.
- Whether an in-memory sort was performed, which indicates an index would be beneficial.
- The number of index entries scanned.
- How long the query took to resolve in milliseconds.

Explain plan will show 0 milliseconds if the query was resolved in less than 1ms, which is not uncommon in well-tuned systems.

When explain plan is called, prior cached query plans are abandoned, and the process of testing multiple indexes is evaluated to ensure the best possible plan is used.

If the application will always use indexes, MongoDB can be configured to throw an error if a query is issued that requires scanning the entire collection.

## Replication Lag

The primary administrative concern that requires monitoring with replica sets, beyond the requirements for any MongoDB instance, is “replication lag.”

In a replica set for a given secondary node, replication lag is the delay between the time an operation occurs on the primary and the time that same operation gets applied on the secondary. Often elevated replication lag is transient and will remedy itself without intervention. Other times, replication lag remains high or continues to rise, indicating a systemic problem that needs to be addressed.

In either case, the larger the replication lag grows and the longer it remains that way, the more exposure your database has to the associated risks such as manual intervention might be needed for reconciling the mismatch or even worst case scenario your system might be forced to run with data which is not up-to-date or it can even be forced to shut down till the primary recovers.

To determine the current replication lag of your replica set, you can use the mongo shell and run the `db.printSlaveReplicationInfo()` command.

```
testset:PRIMARY> db.printSlaveReplicationInfo()
```

Further you can use the `db.printReplicationInfo()` command to fill in the missing piece

```
testset:PRIMARY> db.printReplicationInfo()
```

MongoDB Monitoring Service (MMS) from 10gen can also be used to view recent and historical replication lag. On the Status tab of each SECONDARY node, you'll find the repl lag graph.

Hence in this section we will list down the tips to help reduce this time

**#44:** In scenarios where the write load is heavy, consider having your secondary as powerful as your primary so that it can keep up with the primary. A SECONDARY node should have enough network bandwidth so that it can retrieve ops from the PRIMARY's oplog at roughly the rate they're created and also enough storage throughput that it can apply the ops at that same rate.

**#45:** Adjust the application write concern.

**#46:** If the secondary is used for index builds then it can be planned to be done during a period of low write activity on the primary .

**#47:** If the secondary is used for taking backups then consider taking backups without blocking

**#48:** Check for replication errors. *To check for errors, run rs.status() and look at the errmsg field in the result. Additionally, check the log file of your secondary and look for error messages there.*



One specific example: if you see "RS102 too stale to catch up" in the secondary's `mongodb.log` or in the `errmsg` field when running `rs.status()`, it means that secondary has fallen so far behind that there is not enough history retained by the primary (its "oplog size") to bring it up to date. In this case, your secondary will require a full resynchronization from the primary.

## Sharding

Data is typically sharded when it no longer "fits" on one node. *The goal of sharding is to continually structure a system to ensure data is chunked in small enough units and spread across enough nodes that data operations are not limited by resource constraints.*

**#49** Select a good shard key. In some cases it may not be possible to achieve all goals of sharding with a natural field in the data. It may therefore be required to create a new field for the sake of sharding and to store this in your documents, potentially in the `_id` field. Another alternative is to create a compound shard key by combining multiple fields in the shard key index.

If neither of these options is viable, it may be necessary to make compromises in performance, either for reads or for writes. If reads are more important, then prioritize the shard key selection to the extent the shard key provides query isolation. If writes are more important, then prioritize the shard key selection to the extent the shard key provides write distribution.

**#50** Run three configuration servers to provide redundancy. Production deployments must use three config servers. Config servers should be deployed in a topology that is robust and resilient to a variety of failures.

**#51** Shard collections before you reach 256 GB

# Monitoring

It is important to proactively monitor your MongoDB system for unusual behavior so that actions can be taken to address issues proactively. To monitor your deployment you can use several different tools.

10gen provides a free, hosted monitoring service MongoDB Monitoring Service (MMS) that provides a dashboard and gives you a view of the metrics from your entire cluster. Alternatively you can also build your own tools with nagios, munin or SNMP.

Several tools are provided along with MongoDB that allow you to gain insight into the performance of your deployment such as mongostat, mongotop.

**# 5.2** Using Monitoring services the following should be monitored closely:

1. Op Counters: These include inserts, updates, deletes, reads, and cursor usage.
2. Resident Memory: You should always keep an eye on your memory allocation. Resident memory should always be lower than physical memory. If you go out of memory you'll experience page faults and index misses and have much slower times on query returns.
3. Working set size: Keep a close eye on your working set, which is the total body of data used by your application. For optimal performance, your active working set should fit into RAM. You can decrease the working set size by optimizing your queries and indexing patterns to prevent large scans, or plan to add larger RAM when you expect your working set to increase.
4. Queues: MongoDB's concurrency model uses a readers-writer lock to provide simultaneous reads but exclusive access to a single write operation. Given that approach queues can often form behind a single writer, with those

queues containing readers, writers or both. During lengthy write operations MongoDB will periodically yield to allow readers to get through in order to avoid starvation. Monitoring this metric along with “Lock Percentage” will give you an idea of the concurrency your deployment is seeing. If the “Lock Percentage” and the queues are trending upwards (e.g. spiking) then you may be dealing with contention within the database. Data model changes or “batch” operations can have a significant positive impact on concurrency.

5. Going through your indexing patterns, CRUD behavior and indexes will help you better understand your applications flow for when there is a hiccup.

**# 5.3** Work with full-size databases - Often the performance characteristics of database interactions are dependent on the actual data being manipulated. Hence it's recommended that you do all your performance tests against a copy of your production database because this will avoid many unpleasant surprises later on.

## Summary

In this chapter we provided various How To's which can help the users in forming the journey with MongoDB into a smooth ride.