



WILEY EMERGING TECHNOLOGY SERIES

# Demystifying NoSQL

Seema Acharya

- ▶ Projects in **MongoDB** database inside.
- ▶ Instructive **Videos** to supplement the text.
- ▶ Possible **Interview Questions & Answers** inside.

WILEY



WILEY EMERGING TECHNOLOGY SERIES

# Demystifying NoSQL

Seema Acharya

- ▶ Projects in **MongoDB** database inside.
- ▶ Instructive **Videos** to supplement the text.
- ▶ Possible **Interview Questions & Answers** inside.

WILEY

# Demystifying NoSQL

*OceanofPDF.com*

WILEY EMERGING TECHNOLOGY SERIES

# Demystifying NoSQL

Seema Acharya

Associate Vice President and  
Senior Lead Principal, Infosys Limited

WILEY

*OceanofPDF.com*

# Demystifying NoSQL

Copyright © 2020 by Wiley India Pvt. Ltd., 4436/7, Ansari Road, Daryaganj, New Delhi-110002.

Cover Image: © Nikada/Getty Images

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or scanning without the written permission of the publisher.

**Limits of Liability:** While the publisher and the author have used their best efforts in preparing this book, Wiley and the author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness for any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particular results, and the advice and strategies contained herein may not be suitable for every individual. Neither Wiley India nor the author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

**Disclaimer:** The contents of this book have been checked for accuracy. Since deviations cannot be precluded entirely, Wiley or its author cannot guarantee full agreement. As the book is intended for educational purpose, Wiley or its author shall not be responsible for any errors, omissions or damages arising out of the use of the information contained in the book. This publication is designed to provide accurate and authoritative information with regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services.

**Trademarks:** All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Wiley is not associated with any product or vendor mentioned in this book.

Other Wiley Editorial Offices:

John Wiley & Sons, Inc. 111 River Street, Hoboken, NJ 07030, USA

Wiley-VCH Verlag GmbH, Pappelallee 3, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 1 Fusionopolis Walk #07-01 Solaris, South Tower, Singapore 138628

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario, Canada, M9W 1L1

First Edition: 2020

ISBN: 978-81-265-7996-9

eISBN: 978-81-265-8861-9 (ebk)

[www.wileyindia.com](http://www.wileyindia.com)

Printed at:

*OceanofPDF.com*

*This book is dedicated to my father, who is and will always remain  
my beacon of righteous inspiration.*

*OceanofPDF.com*



# Preface

The last few years have been witness to a burgeoning growth of data. Developers are working with applications that create massive volumes of new, rapidly changing data types – structured, semi-structured, unstructured and polymorphic data. Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users. Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure. Increasingly being used in big data and real-time web applications are NoSQL databases. NoSQL are the “Not only SQL” databases, implying that they support SQL-like query languages, or sit alongside SQL databases in polyglot persistent architectures.

## Need for the Book

We felt the need to compose a book, egged on by the enthusiasm and inquisitiveness of students and instructors alike. A book which can take the readers through an easy comprehension of the world of NoSQL databases. The book is written in simple, easy-to-comprehend language, lots of practical/hands-on examples and exercises to help with self-study and deep dive learning. The book also carries practitioners’ perspective on leveraging the best out of each NoSQL database.

## Audience

This book is for all interested in learning about NoSQL databases. The only criteria is the willingness to learn and the ability to stretch yourself in learning to limits that you have not done before. The book is for all those who are new to NoSQL databases, irrespective of their field/background.

The book will be equally useful to an engineering graduate as it would be to a management graduate. The book has been designed and crafted such that it caters to the knowledge requirements of an IT person as well as a business user.

## Organization of the Book

The book has a total of 6 chapters. Here is a sneak peek into the chapters of the book...

**Chapter 1** introduces you to the world of NoSQL databases, highlights the reasons behind their immense popularity, compares SQL with NoSQL, ACID properties of relational databases with Basically Available Soft State Eventual Consistency (BASE) state of NoSQL databases. The chapter also presents a lucid explanation of the Consistency, Availability and Partition Tolerance (CAP) theorem.

**Chapter 2** introduces you to the various types of NoSQL databases such as key–value, column family, document oriented and graph databases. It discusses the features and advantages of each.

**Chapter 3** focuses on the features of Cassandra, the column family store, and covers the details of Create/insert, Read/select, Update and Delete operations (CRUD). It also explains the import to and export from Cassandra database.

**Chapter 4** dwells on the features of MongoDB, the document store, covers the details of CRUD. It explains the import to and export from MongoDB database. It also explains the aggregation and MapReduce framework in MongoDB.

**Chapter 5** introduces Neo4j, the graph database and discusses its features, benefits, creation of nodes, relationships, indexes, constraints, etc.

**Chapter 6** provides a practitioner's insight to selecting the right NoSQL database for a particular business scenario.

## How to Get the Most Out of This Book

It is easy to leverage the book to gain the maximum by religiously abiding by the following:

1. Read up the chapters thoroughly. Perform hands-on by following the step-by-step instructions stated in demonstrations. Do NOT skip any demonstration. If need be, repeat it a second time or till the time the concept is firmly etched.
2. Explore the various options of all commands.
3. Solve the review questions given at the end of the chapter.
4. Solve the practice exercises provided in the annexure section of the book.

## Next Steps...

We have endeavored to unleash the power of NoSQL databases and introduce you to several scenarios where they have been leveraged successfully. We recommend you read the book from cover to cover, but if you are not that kind of person, we have made an attempt to keep the chapters self-contained so that you can go straight to the topics that interest you most.

Whichever approach you may choose, we wish you well!

## A Quick Word for the Instructor Fraternity

Attention has been paid in arriving at the sequence of chapters and also to the flow of topics within each chapter. This, particularly with an objective to assist our fellow instructors and academicians in carving out a syllabus from the Table of Contents of the book. The complete Table of Contents can qualify as the syllabi for a semester or if the college has an existing syllabus on big data and analytics, a few chapters can be added to the syllabi to make it more robust. We leave it to your discretion on how you wish to use the same for your students.

We have ensured that NoSQL databases discussed in the book is with adequate hands-on content to enable you to teach better and provide ample hands-on practice to your students.

## Instructor Resources

We have also provided the following Instructor Resources that are available for instructors on request. To register, log onto [https://www.wileyindia.com/Instructor\\_Manuals/Register/login.php](https://www.wileyindia.com/Instructor_Manuals/Register/login.php)

1. Chapter-wise Solution Manuals
2. Chapter-wise PowerPoint Presentations (PPTs)

These Instructor Resources are presentation decks (one for each chapter) which can be taken to the class directly or can be customized as per your requirements.

**Happy Learning!!!**

**Seema Acharya**

*OceanofPDF.com*



## About the Author

**Seema Acharya** is an Associate Vice President and Senior Lead Principal with the Education, Training and Assessment department of Infosys Limited. She is a technology evangelist, learning strategist and a passionate tech author with over 20 years of experience in conceptualizing, designing and implementing competency development/enhancement programs in alignment with business strategy to cater to short term as well as long term organizational goals. She collaborates with key stakeholders to study learning needs, craft actionable strategic plans and develop customized learning courses across vast spectrum of technologies such as Business Intelligence, Big Data and Analytics, Open source technologies etc. She is an educator by choice and vocation, and has rich experience in both academia and the software industry.



She is the author of the following books:

1. *Fundamentals of Business Analytics*, ISBN: 978-81-265-3203-2, publisher – Wiley India.
2. *Big Data and Analytics, 2ed*, ISBN: 978-81-265-7951-8, publisher – Wiley India.
3. *Pro Tableau – a step by step guide*, ISBN: 978-14-842-2351-2, publisher – Apress
4. *Data Analytics Using R*, ISBN: 978-93-526-0524-8, publisher – McGraw Hill Higher Education Society

She has co-authored a paper on “Collaborative Engineering Competency Development” for the American Society for Engineering Education (ASEE). She holds the patent on “Method and system for automatically generating questions for a programming language”. Her areas of interest and expertise are centered on business intelligence, big data and analytics technologies such as data warehousing, data mining, data analytics, text mining and data visualization. She is passionate about exploring new paradigms of learning and also dabbles in creating e-learning content to facilitate learning anytime, anywhere.



# Acknowledgements

What an experience it has been! Learning, sharing, exploring, writing, reviewing! This has been the norm for the last few months. I am glad that I undertook this journey. It has been extremely enriching, fulfilling and worth every minute spent in the making. When one undertakes a mammoth task as this, it is never a solo journey. Family, friends and well-wishers join in and make it more memorable. My deepest gratitude to each one involved in the process.

I owe this book to the student and teacher's community who have been readers of my books. They are the reason behind me turning into an author.

I wish to thank my practitioner friends for their good counsel and filling me in on the latest in the field of big data and NoSQL and sharing with me valuable insights on the best practices and technology trends.

I have been fortunate to have the awesome editorial assistance provided by Wiley India. I wish to acknowledge and appreciate Meenakshi Sehrawat, Yoofisaca Nongpluh and their team of associates who adeptly guided me through the entire process of preparation and publication. Thank you for being so patient and putting up with the unpardonable delay in turning in the content.

A special "Thank you" to my family for their constant encouragement and for being my pillars of strength. Thank you for your unconditional support and

love always!

*OceanofPDF.com*



# Contents

[Dedication](#)

[Preface](#)

[About the Author](#)

[Acknowledgements](#)

## Chapter 1    Getting Started with NoSQL

---

[1.1    What has Changed in the Last Decade?](#)

[1.2    History of NoSQL](#)

[1.3    What is NoSQL?](#)

[1.4    Why NoSQL?](#)

[1.5    NoSQL Databases](#)

[1.6    Types of NoSQL Databases](#)

[1.7    SQL versus NoSQL](#)

[1.8    ACID versus BASE](#)

*[1.8.1    ACID](#)*

*[1.8.2    BASE](#)*

[1.9    CAP Theorem](#)

[Remember Me](#)

Question Me  
Reference Me  
Answers

## Chapter 2 Types of NoSQL Databases

---

- 2.1 Introduction
  - 2.2 Key-Value Pair Databases
    - 2.2.1 *What are Keys?*
    - 2.2.2 *What are Values?*
    - 2.2.3 *Differences between Key-Value and Relational Databases*
  - 2.3 Document Databases
    - 2.3.1 *Querying Documents*
    - 2.3.2 *Differences between Document and Relational Databases*
  - 2.4 Column-Family Databases
    - 2.4.1 *Column-Family Database versus Key-Value and Document*
  - 2.5 Graph Database
    - 2.5.1 *Understanding Graph Database*
- Remember Me  
Test Me  
Question Me  
Reference Me  
Answers

## Chapter 3 Column-Family Store

---

- 3.1 Introduction to Apache Cassandra
- 3.2 Features of Cassandra
  - 3.2.1 *Peer to Peer*
  - 3.2.2 *Gossip and Failure Detection*
  - 3.2.3 *Partitioner*
  - 3.2.4 *Replication Factor*
  - 3.2.5 *Writes in Cassandra*
  - 3.2.6 *Hinted Handoff*
  - 3.2.7 *Tunable Consistency*

- 3.3 Cassandra Query Language Data Types
- 3.4 Cassandra Query Language Shell (Cqlsh)
  - 3.4.1 Keyspaces*
- 3.5 Collections
  - 3.5.1 More Practice on Collections (SET and LIST)*
- 3.6 Cassandra Counter Column
- 3.7 Time-to-Live (TTL)
- 3.8 Alter Commands
- 3.9 Import from and Export to CSV
  - 3.9.1 Export to CSV*
  - 3.9.2 Import from CSV*
  - 3.9.3 Import from STDIN*
  - 3.9.4 Export to STDOUT*
- 3.10 Querying System Tables
  - Remember Me
  - Test Me
  - Question Me
  - Practice Me
  - Reference Me
  - Answers

## Chapter 4 MongoDB

---

- 4.1 What is Mongodb?
- 4.2 Why MongoDB?
  - 4.2.1 Using JSON*
- 4.3 Terms used in RDBMS and MongoDB
  - 4.3.1 Document*
  - 4.3.2 Collection*
  - 4.3.3 Embedded Document*
  - 4.3.4 Flexible Schema*
  - 4.3.5 Run MongoDB*
- 4.4 CRUD Operations
  - 4.4.1 Create Database*
  - 4.4.2 Drop Database*

- 4.4.3 *Data Types in MongoDB*
  - 4.4.4 *Arrays*
  - 4.4.5 *Aggregate Function*
  - 4.4.6 *MapReduce Framework*
  - 4.4.7 *Cursors in MongoDB*
  - 4.4.8 *Indexes*
  - 4.4.9 *MongoImport*
  - 4.4.10 *MongoExport*
  - 4.4.11 *Automatic Generation of Unique Numbers for the “\_id” Field*
- Remember Me
- Test Me
- Question Me
- Reference Me
- Answers

## Chapter 5 Neo4j: A Graph-Based Database

---

- 5.1 Introduction to Graph Database
- 5.2 Creating Nodes
  - 5.2.1 *Create a Single Node*
  - 5.2.2 *Verify the Creation of the Node*
  - 5.2.3 *Create Multiple Nodes*
  - 5.2.4 *Create a Node with a Label*
  - 5.2.5 *Create a Node with Multiple Labels*
  - 5.2.6 *Create a Node with Properties*
  - 5.2.7 *Return a Created Node*
- 5.3 Create a Relationship
  - 5.3.1 *Create a Relationship with Label and Properties*
- 5.4 WHERE Clause
  - 5.4.1 *WHERE Clause with Multiple Conditions*
  - 5.4.2 *COUNT() Function*
- 5.5 Creating a Complete Path
- 5.6 Create Index
- 5.7 Create Constraints
- 5.8 Select Data with MATCH

- 5.9 Fetch All Nodes
- 5.10 Drop an Index
- 5.11 Drop a Constraint
- 5.12 Delete a Node
- 5.13 Delete Multiple Nodes
- 5.14 Delete All Nodes
- 5.15 Delete a Relationship
- 5.16 Merge Command

Remember Me  
Question Me  
Practice Me  
Reference Me  
Answers

## Chapter 6 NoSQL Database Orientation

---

- 6.1 RDBMS or NoSQL?
  - 6.1.1 *NoSQL versus SQL*
- 6.2 Key–Value Store
  - 6.2.1 *Pros*
  - 6.2.2 *Limitations*
  - 6.2.3 *Use Cases*
  - 6.2.4 *Few Examples of Key–Value NoSQL Databases*
- 6.3 Column Family Store
- 6.4 Document Store
- 6.5 Graph Store
- 6.6 Examples of NoSQL Databases

6.6.1 *Key–Value Stores*  
6.6.2 *Document Stores*  
6.6.3 *Column Stores*  
6.6.4 *Graph Stores*

Remember Me  
Question Me  
Practice Me  
Reference Me  
Answers

**Annexure A – Project 1 in MongoDB Database**

**Annexure B – Project 2 in MongoDB Database**

**Annexure C – Possible Interview Questions and Answers**

**Index**

*OceanofPDF.com*



# Getting Started with NoSQL

---

## BRIEF CONTENTS

- What has Changed in the Last Decade?
- History of NoSQL
- What is NoSQL?
- Why NoSQL?
- NoSQL Databases
- Types of NoSQL Databases
- SQL versus NoSQL
- ACID versus BASE
  - ACID
  - BASE
- CAP Theorem

---

## 1.1 WHAT HAS CHANGED IN THE LAST DECADE?

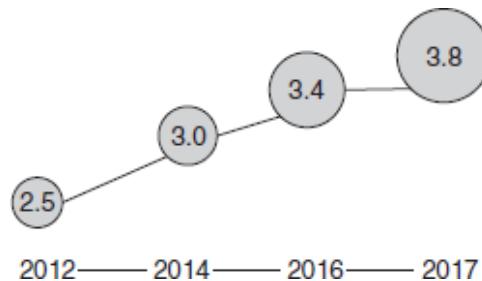
A decade or 10 years may not seem a long time but when it comes to technology it can be a really long time. The number of data sources today cannot be compared to what it was 10 years ago. Data not only comes from the traditional data sources but a lot of it comes from advertising platforms, social media platforms, Customer Relationship Management (CRM) platforms, competitive

intelligence tools, web analytics tools, etc. Today if data is generated, it gets collected and analyzed. The quantity of information captured is far greater than before. Traditional reporting styles have given way to quicker and appealing dashboards. Then there is real-time analytics to garner actionable insights. Below we highlight some of these changes:

**1. Burgeoning Growth of Data:** As per research, 2.5 quintillion bytes (1 quintillion bytes is  $10^{18}$  bytes or 100000000000000000000000 bytes or 1000 petabytes or 1 million terabytes or 1 billion gigabytes) of data gets generated every day. This is at our current pace. Now with Internet of Things (IoT) taking center stage, one can only imagine the accelerated pace at which data is/will be generated.

1 Quintillion bytes = 100000000000000000000000

By 2020, it is predicted that there will be 1.7 MB of data generated per second per person on earth. The global Internet population has grown from 2.5 billion in 2012 to 3.0 in 2014, 3.4 in 2016, 3.8 billion in 2017 (Figure 1.1).



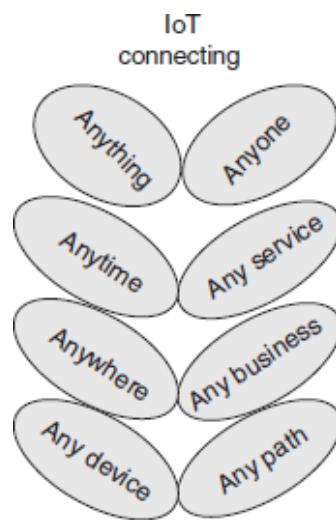
**Figure 1.1** Growth of global Internet population.

Check out the infographic titled “Data Never Sleeps 6.0 – How Much Data is Generated Every Minute” by DOMO to get an idea about the burgeoning growth of data (<https://www.domo.com/learn/data-never-sleeps-6>).

***Some statistics:***

- (a) 120+ professionals join LinkedIn every minute.
- (b) Amazon ships 1,111 packages every minute.
- (c) Google conducts 3,877,140 searches every minute.
- (d) Netflix users stream 97,222 hours of video every minute.

**2. The IoT Surge:** Internet of Things (IoT) can be understood in two parts – “Internet” which is the backbone of connectivity and “Things” which implies computing devices, mechanical and digital machines, objects, persons that are provided with unique identifiers and the ability to send and receive (collect and exchange) data over a network without requiring human-to-human or human-to-computer interaction ([Figure 1.2](#)). Think about the number of smart devices that either you or the people that you interact with use. There are smart phones, smart watches, smart home appliances, smart office appliances, smart transportation management systems, smart weather forecasting systems, smart cities, etc. What this means is that there will be more data – not just data that is humungous in volume but data that is rich in variety (structured, semi-structured, and unstructured data).



**Figure 1.2** Internet of Things (IoT).

**3. Need for Real-Time and Deeper Analytics:** There was a time when managers and business users were happy with the analysis that you ran on historical data, but times have changed. Today, there are more expectations from the data. There is a need to analyze data in real time. Real-time analytics implies that analysis of data takes place or should take place as soon as the data becomes available. There is a need for businesses to react and respond without delay. To surge ahead of competition, there is a constant need to gain immediate insights and draw inferences from data. It allows opportunities to be cashed in time and problems to be averted before they snowball.

**4. Era of Polyglot Persistence:** There was a time when an application had to be written in one programming language. We have realized through experience that different programming languages are best suited for different problem types. The term “Polyglot” was coined by Neal Ford in 2006 to refer to the usage of different programming languages to address different purposes in an application. Since then there has been no looking back. Just like an application can use multiple programming languages to address the specific problems at hand, likewise different data stores can be used for different types of data, basis the usage of data by the application or application component.

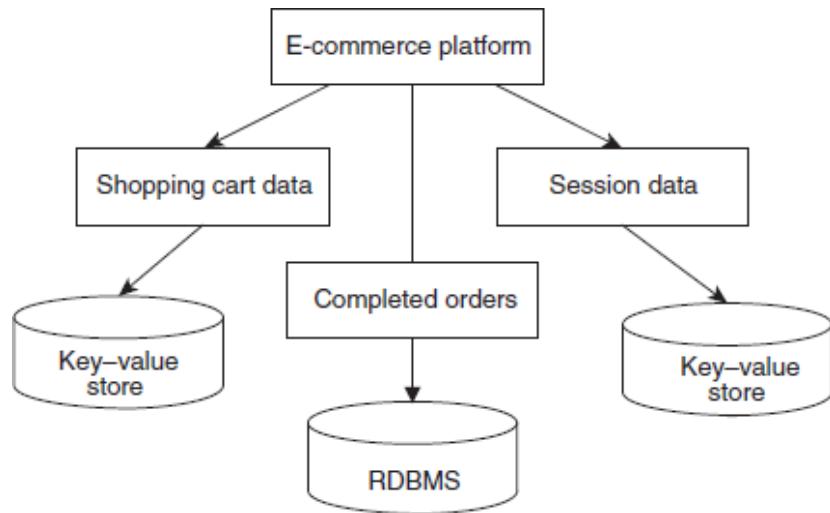
---

### Picture This

You are a developer working on an e-commerce application. The e-commerce application will provide the following functionalities to the customers:

- (a) To browse for products/items as per their specifications (such as clothing from a particular brand, apparels of a particular size, type, mobile phone of particular specifications, etc.).
- (b) To place products into the wish list to be viewed later.
- (c) To place products into the shopping cart (products that they intend to buy).
- (d) Perform the transaction (checkout the products from the shopping cart by making the payment).
- (e) Get recommendations based on their buying patterns, etc.

Will a single database work for this application? Does it look like an ideal scenario for polyglot persistence? A key-value data store could be used to store the shopping cart data before the order is confirmed by the customer. Key-value pair database can also be used to store the session data. The requirement here is to hold transient data, support for frequent reads and writes and simpler queries using the unique keys. Key-value stores make for an ideal choice since the shopping cart is usually accessed by user ID and session data is keyed by the session ID. Once the order is confirmed and paid by the customer, it can be saved in the RDBMS ([Figure 1.3](#)).



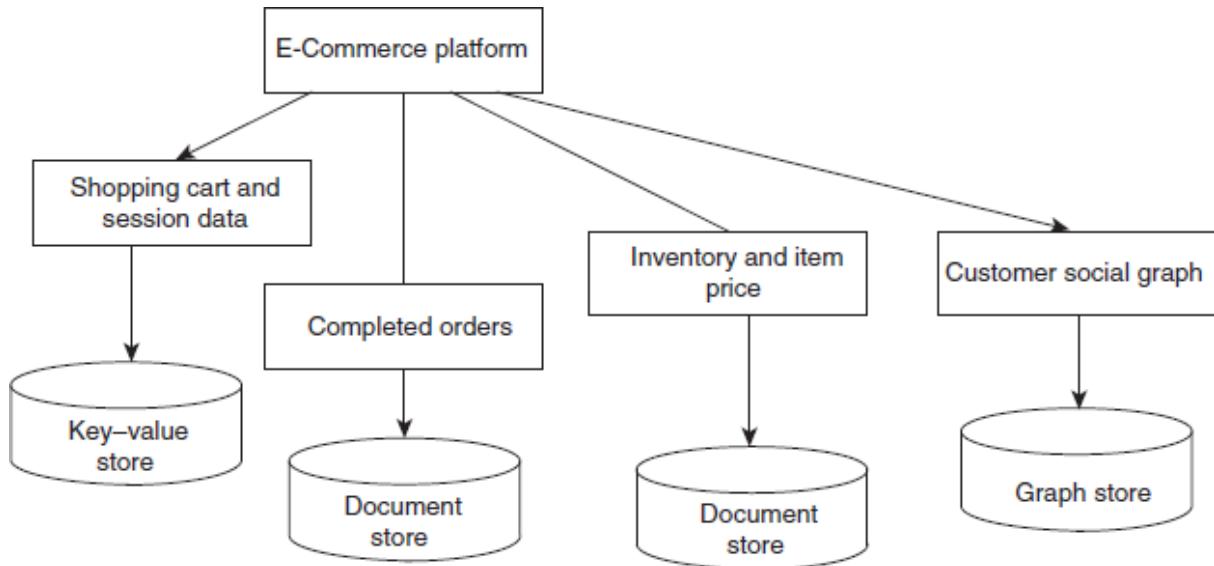
**Figure 1.3** Use of key-value stores to offload session and shopping cart data storage.

However, to recommend products to customers when they place products into their shopping carts – for example, “customers who bought these products also bought these” or “customers who viewed these products also viewed these products” – a graph data store fits the bill perfectly. The graph data store is the most appropriate to represent relationship(s) between data (Table 1.1).

**Table 1.1** Different NoSQL databases for different applications

Type of Data/Application	The Right NoSQL Database
Shopping cart and session data	Key-value store
Completed orders/transactions	Relational databases/document stores
Product catalogs/inventory	Document store
Customer social graph (interconnections among customers)	Graph store
Recommendation engine (what should/can you buy along with the products that you bought)	Graph store

Even using specialized relational databases for different purposes, such as data warehousing appliances or analytics appliances within the same application, can be viewed as polyglot persistence (Figure 1.4).



**Figure 1.4** Example implementation of polyglot persistence.

## 1.2 HISTORY OF NoSQL

---

The term NoSQL was first coined by Carlo Strozzi in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface. Johan Oskarsson, who was then a developer at last.fm, in 2009 reintroduced the term NoSQL at an event called to discuss open-source distributed network. The hashtag, “#NoSQL” was coined by Eric Evans, and few other database persons at the event found it suitable to describe these non-relational databases. A few features of NoSQL databases are:

1. They are open source.
2. They are non-relational.
3. They are distributed.
4. They are schema-less.
5. They are cluster friendly.
6. They are born out of 21<sup>st</sup> century web applications.

## 1.3 WHAT IS NoSQL?

---

A NoSQL database is “Non SQL”, “Non-relational”, “Not Just SQL” and mostly open source. It is an alternative to traditional relational databases. The changing scenarios such as the ones listed below have led to the advent of the NoSQL databases:

1. Today developers work with applications that create massive volumes of new, rapidly changing structured, semi-structured, unstructured, and polymorphic data. Think about Netflix, Amazon, etc.
2. Long gone is the 12-to-18 month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
3. Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.
4. Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

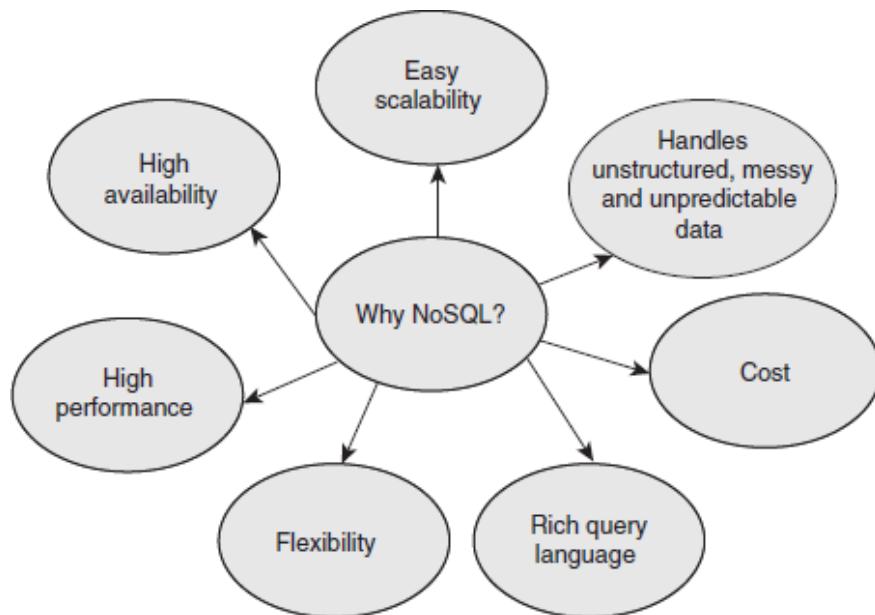
## 1.4 WHY NoSQL?

---

The benefits of using NoSQL are multifold (Figure 1.5). We explain some of these below:

1. **Easy Scalability:** What is scalability? Scalability is the ability to efficiently deal with varying workloads.

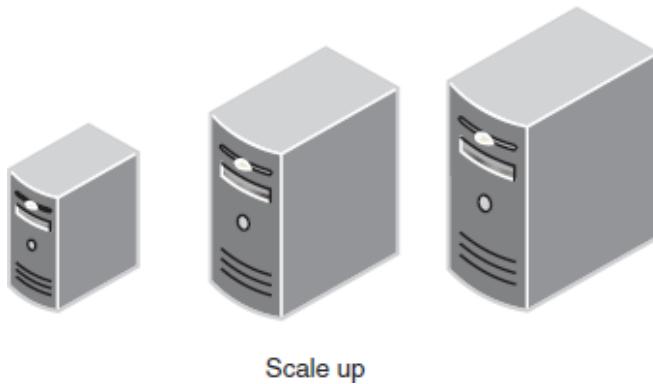
**Scalability = Ability of the system to handle growing amount of work**



**Figure 1.5** Benefits of NoSQL.

## Picture This

You have a traditional RDBMS setup. The number of concurrent users has increased significantly. The load on the server has increased. Performance is becoming an issue. It is time to upgrade the database server. You begin by increasing the horse power of the server by adding more processors, RAM, hard disks, increased network bandwidth, etc. In [Figure 1.6](#), we have depicted just ONE server and NOT three. The server is being gradually ramped/scaled up by adding extra resources such as RAM, hard disks, etc.



**Figure 1.6** Scale up or vertical scalability.

However, there is a limit to maxing out the server. Another option could be to replace the existing server with a higher end one with more CPUs, memory, and so on. Barring the cost that will be incurred to scale up, there is yet another flip side. If the scale up is achieved by replacing an existing server, it will require the database management system to be migrated to new higher configuration server. And if the scale up has been achieved by adding resources, it would not need a migration of the database management system; however, it would require some downtime to upgrade the hardware.

### **Vertical Scalability or Scale Up = Handle More Load by using Faster Processors**

Also there is a high chance of under-utilizing the server during the “less than peak load”.

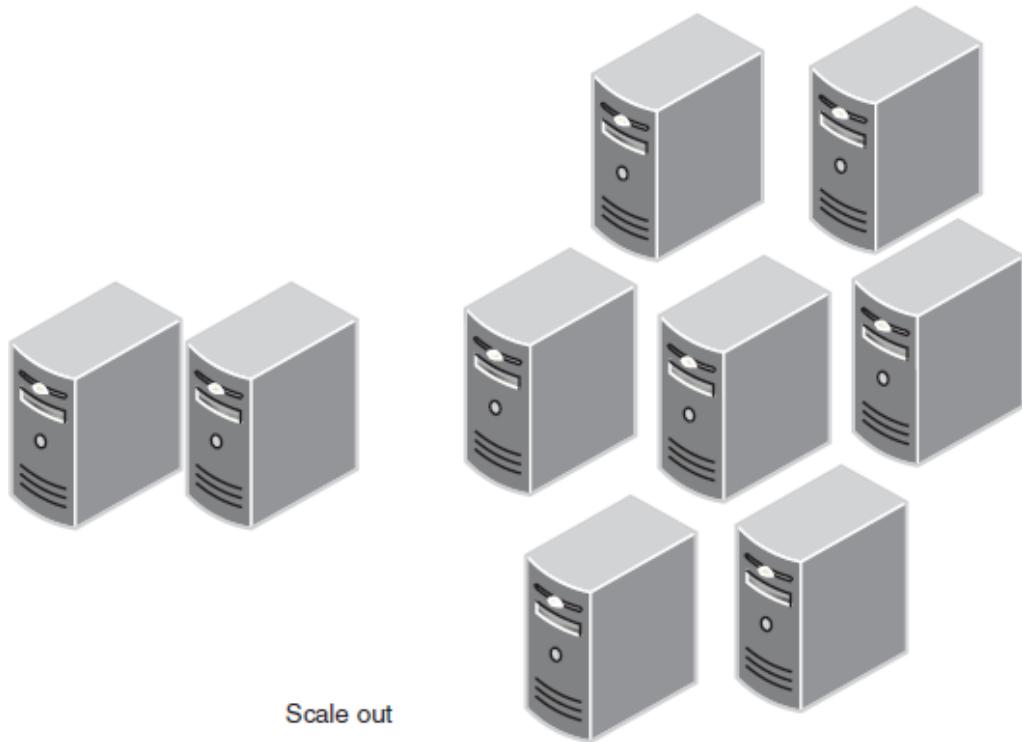
We would benefit most by a setup that allows additional servers to be added when we hit peak load or experience spikes and to shut down additional servers when the spike subsides. Adding servers as and when required is called scaling out.

Can scaling out be achieved with RDBMS? Yes, however this may need additional software to manage the multiple servers that function as a single database system. Also this could lead to increased complexity, not to forget the additional operational costs.

The limitations stated above can be overcome by NoSQL owing to its scaling out trait. Scaling out or linear scaling allows the load to be distributed to multiple nodes once the maximum capacity of your current application or database server is reached. Refer [Figure 1.7](#). Also the scaling out can be performed in run time. There is no need to stall operations and it is a less expensive solution owing to use of commodity hardware. Most of the NoSQL database management systems are designed to adjust automatically or through minimal database administrator intervention to the addition or removal of servers to the cluster.

### **Horizontal Scalability or Scale Out = Handle More Load by using More Processors**

- 2. High Performance:** NoSQL is a good choice for companies experiencing rapid growth with no clear schema definitions. NoSQL offers much more flexibility than a relational database and is a solid option for companies that analyze large quantities of data or manage variable data structures.
- 3. High Availability:** This implies continuous uptime (100% uptime). It also implies that there will be ample support for backup and failover. Failover is an important fault-tolerance function. When the primary database is unavailable either because of failure or scheduled downtime, failover ensures that secondary system components/backup systems assume the functions of the primary component. It often happens automatically and transparently. Some “distributed” NoSQL databases use a master-less architecture that automatically distributes data equally among multiple resources so that the application remains available for both read and write operations in the event of hardware or network failures. For example, CouchBase has a master-less architecture vis-a-vis MongoDB which has a master-slave architecture. Maximum resiliency is guaranteed with a master-less distributed NoSQL as the data gets evenly distributed across several nodes in the cluster. There are multiple redundancies built into the system by default which prevents single points of failures and downtimes.



**Figure 1.7** Scale out or horizontal scalability.

4. **Cost:** Cost is an important consideration when it comes to zeroing in on a database or databases (remember polyglot persistence). Commercial database software vendors have several licensing models. Few are listed below for easy comprehension:
  - (a) Basis the number of concurrent users.
  - (b) Basis the number of named users.
  - (c) Basis the size of the database server.

It is typically difficult to comment on the utilization of database servers. As an example consider Web applications. They may experience spikes where the number of user access to the database servers swells. Likewise, there could be periods of underutilization as the spikes recede. What is a good measure? What should be taken into consideration? The number of peak users or the number of average users. What is advisable? Term licenses or perpetual licenses. What criteria to use to accurately predict the number of users six months or one year down the line?

Contrast this to open-source NoSQL databases. Majority of the NoSQL databases are available as open source – no licensing hassles, etc. There are third-party vendors who provide commercial support services so the

NoSQL users do not miss out on support received by proprietary RDBMS users.

**5. Flexibility:** Let us take a moment to think back on the RDBMS that we are familiar with. With RDBMS, the limitation is that one has to think, design, and define the schema much before the data is pushed into the tables of the database. It is assumed that largely all the rows will have the same columns.

**Example 1:** A products table. Think about all the attributes, properties of a product that you would like to collect data about. Design and define the table schema and then insert the data. It does not take into consideration the fact that different products will have different attributes.

For instance, a nutriblend complete kitchen machine will have very different attributes compared to those of an apparel which will have very different attributes from those of a book. We are left with two choices:

- (a) Design a table with all the attributes possible for all the products. There will be quite a few columns which will be relevant for a product but irrelevant for quite a few.
- (b) Create a separate table for each product.

NutriblendCKM							
MotorPower	OperatingVoltage	JarType	NoofBlades	AdditionalCaps	Accessories	Price	
Apparel							
Brand	Fabric	Style	SleeveType	NeckType	Size	Price	
Book							
BookTitle	AuthorName	Publisher	Type	KindleEdition	YearofPublication	Price	ISBNNo
							NoofPages

Both the options have their pros and cons. In option (a) it leads to a sparse table. In the second, it will lead to performance overhead whenever it is required to fetch information for more than one product.

All these challenges can be addressed by NoSQL databases such as MongoDB. A MongoDB collection can have documents, each with different set of key-value pairs.

```

Product
{
{
"ProductName": "NutriblendCKM",
"MotorPower": "22000 RPM",
"OperatingVoltage": "400 Watts",
"JarType": "Big Jar", "Juicer", "Blender", "Chopper",
"NoofBlades": 3,
"AdditionalCaps": 2,
"Accessories": "Seasoning Cap",
"Price": "3790 INR"
}
{
"ProductName": "GAP Tshirt",
"Brand": "GAP",
"Fabric": "100% Cotton",
"Style": "Polo",
"SleeveType": "Hemmed Sleeves",
"NeckType": "Polo",
"Size": "L",
"Price": "1100 INR"
}
{
"ProductName": "Book",
"BookTitle": "Data Analysis using R",
"AuthorName": "Seema Acharya",
"Publisher": "Mcgraw Hill Higher Education Society",
"Type": "Paper back",
"Kindle Edition": "Yes",
"YearofPublication": 2018,
"Price": 545,
"ISBNNo": 9945678912,
"NoofPages": 580
}
}

```

Most of the NoSQL databases do not require the schema to be pre-defined. In other words, they do not have any requirement of a fixed table structure.

6. NoSQL handles (manage and store) unstructured, messy, and unpredictable data. Imagine the kind of data that is up on e-commerce site. All e-commerce applications maintain a product catalog. There is description available for each product in the product catalog besides other information such as price/cost, etc. The description may include images/photographs of the product. Some products may have a video demonstrating how to use it. Images, videos, etc. constitute unstructured data. The network traffic to the e-commerce application can become unpredictable especially on “big sale

day". NoSQL database helps to manage unpredictable volume of data that most often is unstructured.

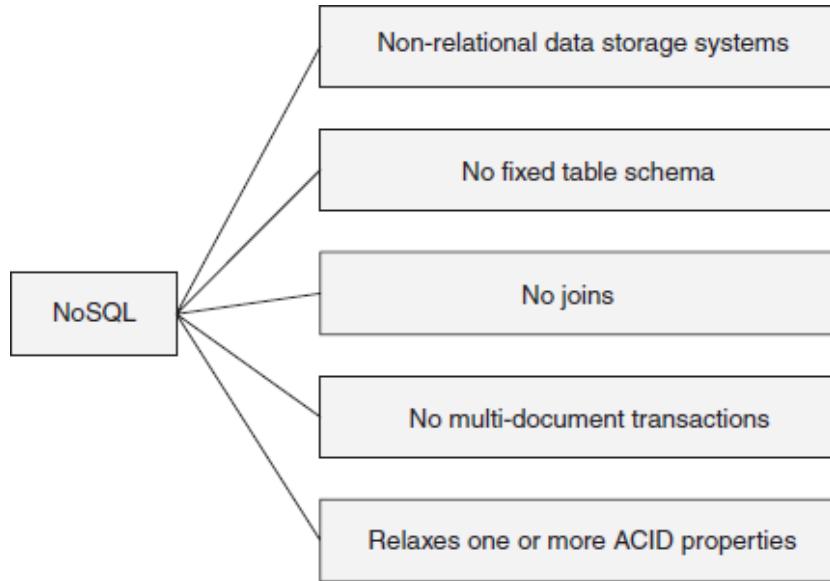
7. **Availability:** What do you do when your favorite social media site such as Facebook, Twitter, etc. is frequently down? What do you do when your favorite e-commerce site hangs most of the time? You begin scouting for a new favorite. Let us understand availability in the context of single server and multiple server databases.
  - (a) When the database is on a single server machine and if that machine fails, the complete work comes to a standstill with the application becoming unavailable.
  - (b) When the database is on a single server machine and there is a backup server for this primary server. The Backup server maintains replicated copies of the data from the primary server. If the primary server fails, the backup server springs into action. There may be a time lag as the backup server assumes the responsibility previously shouldered by the primary server. Moreover, the backup server is used only in the event of a failure and otherwise does not assist in partaking of the load from the primary server.
  - (c) Database is distributed on multiple commodity (low cost) servers. If one of the server goes down for some reason, the workload is taken over by other servers in the cluster.
8. NoSQL is a rich query language. NoSQL databases such as MongoDB supports a full query language, primary and secondary indexing, full text search, etc.

### TRY THIS – 1

Which of the following statements relating to SQL and NoSQL are true?

- (a) SQL has a relational model.
- (b) SQL supports horizontal scalability.
- (c) NoSQL is meant only for semi-structured and unstructured data whereas SQL is used for structured data.
- (d) NoSQL databases can have flexible schema.

## 1.5 NoSQL DATABASES



**Figure 1.8** What is NoSQL?

NoSQL databases are non-relational, open-source, distributed databases ([Figure 1.8](#)). They are hugely popular today owing to their ability to scale out or scale horizontally and the adeptness at dealing with a rich variety of data: structured, semi-structured and unstructured data. NoSQL databases

- 1. Are Non-Relational:** They do not adhere to relational data model. In fact they are either key–value pairs or document-oriented or column-oriented or graph-based databases.
- 2. Are Distributed:** They are distributed meaning the data is distributed across several nodes in a cluster.
- 3. Provide No Support for ACID Properties (Atomicity, Consistency, Isolation, and Durability):** They do not offer support for ACID properties of transactions. On the contrary, they adhere to Brewer's CAP (Consistency, Availability and Partition tolerance) theorem and are often seen compromising on consistency in favor of availability and partition tolerance.

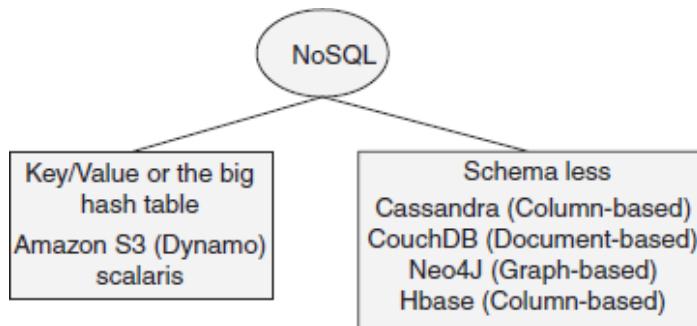
NoSQL databases are widely used in big data and other real-time web applications.

## 1.6 TYPES OF NoSQL DATABASES

We have already stated that NoSQL databases are non-relational. They can be broadly classified into the following:

1. Key-value or the big hash table.
2. Schema-less.

Refer [Figure 1.9](#). Let us take a closer look at key-value and few other types of schema-less databases:



**Figure 1.9** Types of NoSQL databases.

1. **Document:** It maintains data in collections constituted of documents. For example, MongoDB, Apache CouchDB, Couchbase, MarkLogic, etc.

***Sample Document in Document Database:***

```
{
  "Book Name": "Fundamentals of Business Analytics",
  "Publisher": "Wiley India",
  "Year of Publication": "2011"
}
```

2. **Column:** In a column-oriented data store, data is stored in cells which are grouped into columns of data rather than rows. Columns that are logically related are grouped to constitute column families. A column family can contain virtually unlimited columns of data which can be created at runtime or defined during the design of schema. Read and write is performed using columns rather than rows. Columns in a column family are usually accessed together. For example, the customer name is usually accessed along with his/her profile information. The order details are not generally fetched alongside the customer name. Examples of column-oriented NoSQL databases are Cassandra, Hbase, etc. Refer [Figure 1.10](#). A row in the column family database is a collection of several columns. A row is identified by a unique Row Key. Each column has a name/key, the value contained, and a timestamp which helps to know the date and time that the value was updated/modified.

Row	Row Key	Column 1	Column 2	Column <i>n</i>
		Name	Name	Name
		Value	Value	Value
		Timestamp	Timestamp	Timestamp

**Figure 1.10** Sample column data store.

3. **Key–Value:** It maintains a big hash table of keys and values. For example, Dynamo, Redis, Riak, etc.

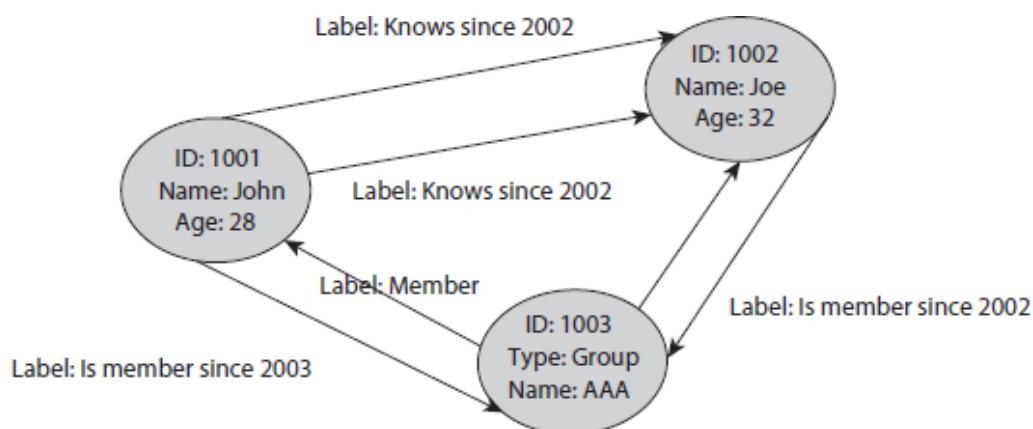
***Sample Key–Value Pair in Key–Value Database:***

Key	Value
First Name	Simmonds
Last Name	David

4. **Graph:** They are also called network databases. They store data in nodes. For example, Neo4j, HyperGraphDB, etc.

***Sample Graph in Graph Database:***

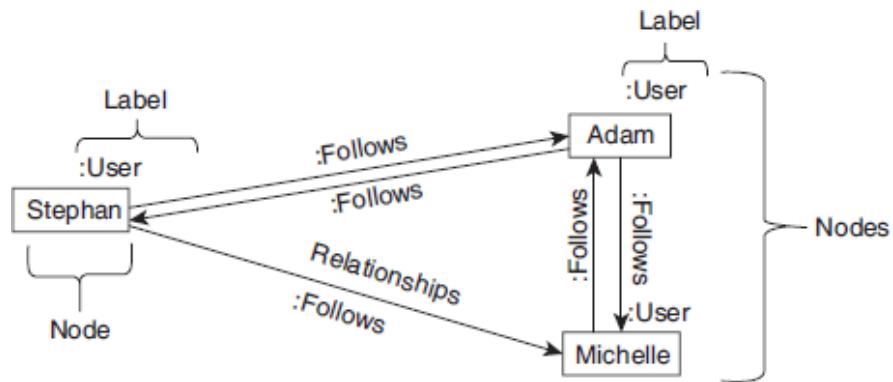
**Example 1:** John and Joe know each other since 2002. Joe is a member of a group “AAA” since 2002. It was only in 2003 that John too became a member of the group, “AAA”. Refer [Figure 1.11](#).



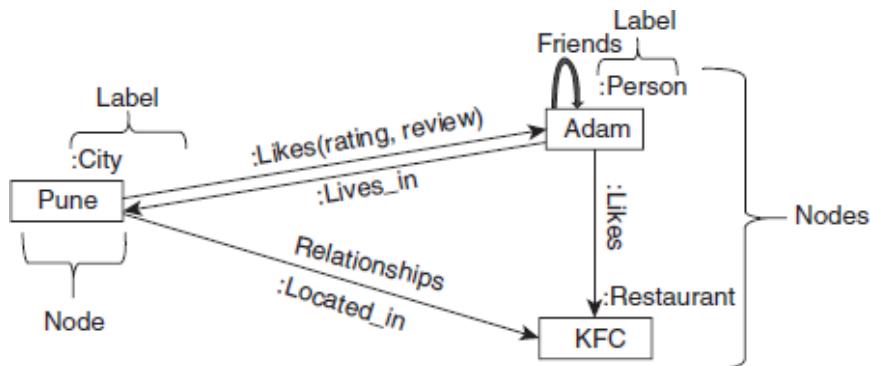
**Figure 1.11** A graph depicting relationship between members of a social group.

**Example 2:** On a social networking site, Stephan follows Adam. Adam too follows Stephan. Likewise, Adam and Michelle follow each other. Stephan also follows Michelle. However, Michelle is yet to follow Stephan. Refer Figure 1.12.

**Example 3:** Adam lives in the city “Pune”. He likes to eat out at the restaurant “KFC” located in “Pune” city. Refer Figure 1.13.



**Figure 1.12** A graph depicting a social networking site.



**Figure 1.13** A graph depicting relationship between entities Person, City, and Restaurant.

## 1.7 SQL VERSUS NoSQL

Table 1.2 gives a few differences between SQL and NoSQL.

**Table 1.2** SQL versus NoSQL

	SQL	NoSQL
Type	Relational database	Non-relational
Architecture	Centralized	Distributed
Data model	Relational model	Model-less approach
Flexibility	Every record conforms to a pre-defined schema. Columns must therefore be determined and locked before entering data. It can ONLY be amended later by taking the database offline and then modifying the entire database.	Dynamic schema for unstructured data. Dynamic schema allows information to be updated on the fly.
Data store type	Table-based databases. Data is stored in rows and columns.	Document-based or graph-based or wide column store or key-value pairs databases.
Scalability	Vertically scalable (by increasing the horse power of the machine – add system resources such as CPU, disk memory, RAM, etc.).	Horizontally scalable (by creating a cluster of low cost, easily available commodity machines).
Query language	Uses Structured Query Language (SQL).	Uses Unstructured Query Language (UnQL).
Suitability	Not preferred for large datasets.	Largely preferred for large datasets/big datasets.
Support for hierarchical storage	Not a best fit for hierarchical data.	Best fit for hierarchical storage as it follows the key-value pair of storing data similar to JSON (Java Script Object Notation).
ACID compliance	Adheres to ACID (Atomicity, Consistency, Isolation and Durability) properties.	Follows BASE (Basically Available Soft State Eventual Consistency) properties.
Vendor support	Excellent support from vendors.	Relies heavily on community support.
Usage	Supports complex querying and data keeping needs.	Does not have good support for complex querying.
Consistency	Can be configured for strong consistency.	Few support strong consistency (e.g., MongoDB), few others can be configured for eventual consistency (e.g., Cassandra).
Cost	Expensive high-end servers – mostly proprietary software with additional licensing cost.	Open-source software on low-cost commodity machines reduce licensing costs by 50–80%.
Performance	Performance degrades as concurrent users increase.	Multiple nodes to process multiple requests.
Latency	On-disk processing slows down query performance.	In-memory caching processing delivers sub-millisecond response.
Reliability	Request cannot be processed if the central server is down.	Data is copied to multiple nodes overcoming single node failure.
When?	(a) High volume of transactional data. (b) Data is structured.	(a) Requires faster processing of real-time data. (b) No requirement to protect transactional data.
Examples	Oracle, DB2, MySQL, MS SQL, PostgreSQL, etc.	MongoDB, HBase, Cassandra, Redis, Neo4j, CouchDB, Couchbase, Riak, etc.

## TRY THIS – 2

---

Place them in the correct basket, “SQL” or “NoSQL”.

SQL	NoSQL

- (a) Relational
- (b) Distributed
- (c) Predefined schema
- (d) Wide-column stores
- (e) Vertically scalable
- (f) Key–value pairs
- (g) MySQL
- (h) CouchDB
- (i) Neo4j
- (j) Cassandra
- (k) Large dataset
- (l) ACID properties
- (m) Brewers CAP theorem
- (n) Document based database
- (o) Scales horizontally
- (p) Avoids join operations
- (q) JSON data
- (r) Table or relations

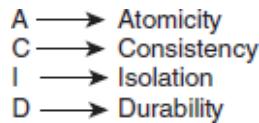
---

## 1.8 ACID versus BASE

---

### 1.8.1 ACID

What does the term ACID mean?



Now let us explain these terms individually:

**1. A – Atomicity:** The word atomicity is derived from the word “Atomos” meaning it is indivisible. It imposes the ALL or NONE principle. Either all or none of the tasks in a transaction are performed.

**Example:** Alex is transferring 1000 USD from his Account A to his Account B. This fund transfer involves the following tasks:

- (a) Debit 1000 USD from Account A
- (b) Update Account A
- (c) Credit 1000 USD to Account B
- (d) Update Account B

Atomicity implies that either all the four tasks will be performed or none will be performed. It cannot happen that 1000 USD is debited from Account A but is not credited to Account B.

**2. C – Consistency:** The database should be in a consistent state before as well as after a transaction.

Consider again, the fund transfer of 1000 USD from Account A to Account B. Before the transaction, assume that the balance in Account A and Account B was 5000 USD and 2000 USD, respectively.

Account A → 5000 USD  
Account B → 2000 USD

Total Amount (summing up the balance in Account A and Account B) =  $5000 + 2000 = 7000$  USD

After the transaction, the account balance will be:

Account A → 4000 USD  
Account B → 3000 USD

Total Amount (summing up the balance in Account A and Account B) =  $4000 + 3000 = 7000$  USD.

Refer [Figure 1.14](#). It cannot happen that Account A has been debited of 1000 USD but Account B has not been credited of 1000 USD.

Account A =  $5000 - 1000 = 4000$  USD  
Account B = 2000 USD

Before	Account A = 5000	Account B = 2000
Transactions		
T1		T2
Read (Account A)		Read (Account B)
Account A = Account A - 1000	Account B = Account B + 1000	
Write (Account A)		Write (Account B)
After	Account A = 4000	Account B = 3000

**Figure 1.14** Sample transaction – transfer of funds from account A to Account B.

Total Amount (summing up the balance in Account A and Account B) =  $4000 + 2000 = 6000$  USD

This is inconsistent data. RDBMS does not allow this.

3. ***I – Isolation:*** No transaction has access to any other transaction that is in an intermediate or in an unfinished state. Thus, each transaction is independent unto itself. This ensures both performance and consistency of transactions within a database.

In the above example of fund transfer, a transaction is isolated from reading Account A or Account B till both the accounts are updated.

4. ***D – Durability:*** Once the transaction is complete, it will persist as complete and cannot be undone; it will survive system failure, power loss, and other types of system breakdowns.

*Summarizing...*

1. **Where is ACID used?** It is used with transactions supported by RDBMS.
2. **Why is it used?** To achieve high consistency.
3. **How is it achieved?** During a transaction, the database moves from one steady or consistent state to another steady or consistent state. The transaction is atomic which means that either the transaction is committed in its entirety or none of it is performed at all.

### 1.8.2 BASE

What does the term BASE mean?

B	→	Basically
A	→	Available
S	→	Soft state
E	→	Eventually consistent

Now let us explain these terms individually:

**1. BA – *Basically Available*:** Assume that you have a distributed NoSQL setup. This implies that the NoSQL database is running on multiple servers (let us assume 10 servers). There are two situations:

- (a) NoSQL database is running on 10 servers without replication, that is, no piece of data is duplicated or replicated. Assume one of the server fails. What is likely to happen to the queries fired by you?
  - i. All processing will come to a standstill because a server from amongst the 10 has failed.
  - ii. 10% of the users' queries will fail but 90% of the queries will succeed.

The correct answer here is the second one (ii). This is because NoSQL database is distributed over 10 servers and out of the 10 servers, only one server has crashed. Assuming that each of the server is equally utilized only 10% of the queries fired will fail.

- (b) NoSQL database is running on 10 servers with replication, that is, NoSQL database has kept multiple copies of the data on different servers. Assume one or more server has failed. What is likely to happen to the queries fired by you?
  - i. All processing will come to a standstill because a server or more than one server from amongst the 10 has failed.
  - ii. 10% (if one server fails), 20% (if two servers fail), etc. of the users' queries will fail but 90% queries in the first case and 20% queries in the second case will succeed.
  - iii. All the queries will get a response; however, with a little latency. The latency will depend on how many servers have crashed.

The correct answer is the third option (iii). Basically available implies that there can be a partial failure in some parts of the distributed system but the rest of the system continues to function taking portions of the load of the failed system.

**2. S – *Soft State*:** What is the meaning of soft state? Strictly going by computer science, soft state implies that data will expire if it is not refreshed. However in NoSQL database, soft state implies that the data may eventually be overwritten with more recent data. This property from the

BASE transaction is not to be considered in silos as there is an overlap with the third property, “eventually consistent”.

3. ***E – Eventually Consistent:*** This means that there may be intermittent periods of inconsistency for the database but eventually the database will be placed in a consistent state. NoSQL databases keep multiple copies of the data on multiple servers. Assume that you are working with a piece of data that has three copies placed on three different servers. You update one copy. The other two copies still have the old version of the data. This is a temporary state of inconsistency. The replication mechanism will eventually update all the copies and the database will be consistent as before.

One could experience a time lag as updates are made to multiple copies of the data. This time lag may depend on the load on the system, the number of replicas or copies to update, and the speed of the network. For example, users’ profile information is stored in the NoSQL database. There are three copies of the same data on three different servers. Two servers are on the same local area network while the third server is in a datacenter thousands of miles away. The user updates his/her profile information on one of the servers that shares the local area network with another server. The update to the copy on this other server happens quickly. However, to update the copy on the server sitting several thousand miles away, it takes a while. The user querying server 1 or 2 will get the recent updates but if the querying takes place as the updates are in progress, the user querying the 3<sup>rd</sup> server still sees the old information.

### ***Summarizing...***

1. **Where is BASE used?** In distributed computing.
2. **Why is it used?** To achieve high availability.
3. **How is it achieved?** Assume a given data item. If no new updates are made to this given data item for a stipulated period of time, eventually all accesses to this data item will return the updated value. In other words, if no new updates are made to a given data item for a stipulated period of time, all updates that were made in the past and not applied to this given data item and the several replicas of it will percolate to this data item so that it stays as current/recent as is possible.
4. **What is replica convergence?** A system that has achieved eventual consistency is said to have converged or achieved replica convergence.

How is the conflict resolved?

1. **Read repair:** If the read leads to discrepancy or inconsistency, a correction is initiated. It slows down the read operation.

2. **Write repair:** If the write leads to discrepancy or inconsistency, a correction is initiated. This will cause the write operation to slow down.
3. **Asynchronous repair:** Here, the correction is not part of a read or write operation.

## 1.9 CAP THEOREM

---

The CAP theorem is also called as the Brewer's theorem. It states that in a distributed computing environment (a collection of interconnected nodes that share data), it is impossible to provide the following guarantees. At best you can have two of the following three – one must be sacrificed.

1. **Consistency** implies that every read fetches the last write.
2. **Availability** implies that reads and writes always succeed. In other words, each non-failing node will return a response in a reasonable amount of time.
3. **Partition tolerance** implies that the system will continue to function when network partition occurs.

Let us understand this using a real-life situation.

You work for a training institute “XYZ”. The institute has 50 instructors including you. All of you report into a training coordinator. At the end of the month, all the instructors together with the training coordinator look through the training requests received from the various corporate houses and prepare a training schedule for each instructor. These training schedules (one for each instructor) are shared with “Amey”, the office admin. Each morning you either call the office helpdesk (essentially Amey's desk) or check in-person with Amey for your schedule for the day. In case a training request has been cancelled or updated (updates can be in the form of change in course, change in duration, change of the training timings, etc.) Amey is informed of the updates and the schedules are subsequently updated by him.

Things were good until now. Few corporate houses were your client and the schedules of each instructor could be smoothly managed without any major hiccups. But your training institute has been implementing promotion campaigns to grow the business. As a consequence of advertising in the media and word of mouth publicity by your existing clients, you suddenly see an upsurge in training requests from existing and new clients. Consequence of that more instructors have been recruited. Few trainers/consultants have also been roped in from other training institutes to help tackle the load.

Now when you go to Amey to check your schedule or call in at the helpdesk, you are prepared for a wait in the queue. Looking at the current state of affairs, the training coordinator decides to recruit an additional office admin, "Joey". The helpdesk number will remain the same and will be shared by both the office admins.

This arrangement works well for a couple of days. Then one day...

*You:* Hey Amey!

*Amey:* Hi! How can I help?

*You:* I think I am scheduled to anchor a training at 3:00 pm today. Can I please have the details?

*Amey:* Sure! Just a minute. Amey browses through the file where he maintains the schedules. He does not see a training scheduled against your name at 3:00 pm today and responds back, "you do not have any training to conduct at 3:00 pm".

*You:* How is that possible? The training coordinator called up yesterday evening to inform of the same and said he has updated the office admins of the same.

*Amey:* Oh! Did he say which office admin? It could have been Joey. Please check with Joey.

*Amey:* Hey Joey! Please check the schedule for Paul here... Do you see something scheduled at 3:00 pm today?

*Joey:* Sure enough! He is anchoring the training for client "Z" today at 3:00 pm.

***A clear case of inconsistent system!!!*** The updates in the schedule were shared by the training coordinator with Joey and you were checking for your schedule with Amey.

You share this incident with the training coordinator and that gets him thinking. The issue has to be addressed immediately otherwise it will be difficult to avoid a chaotic situation. He comes up with a plan and shares it with both the office admins the following day.

*Training Coordinator:* Folks, each time that either I or an instructor calls any one of you to update a schedule, make sure that both of you update it in your respective files. This way the instructor will always get the most recent and consistent information irrespective of whom amongst the two of you they speak to.

*Joey:* But that could mean a delay in answering either a phone call or sharing the schedule with the instructor waiting in queue.

*Training Coordinator:* Yes, I understand but there is no way that we can give incorrect information.

*Amey:* There is this other problem as well. Suppose one of us is on leave on a particular day. That would mean that we cannot take any update calls as we will not be able to simultaneously update both the files (my file and Joey's).

*Training Coordinator:* Well, good point! ***That's the availability problem!!!*** But I have thought about that as well. Here is the plan:

- (a) If one of you receive the update call (any updates to any schedule), ensure that you inform the other person if he is available.
- (b) In case the other person is not available, ensure that you inform him of all the updates to all schedules via email. It is a must!!!
- (c) When the other person resumes duty, the first thing he will do is to update his file with all the updates to all schedules that he has received via email.

Wow!!! That is sure a Consistent and Available system!!!

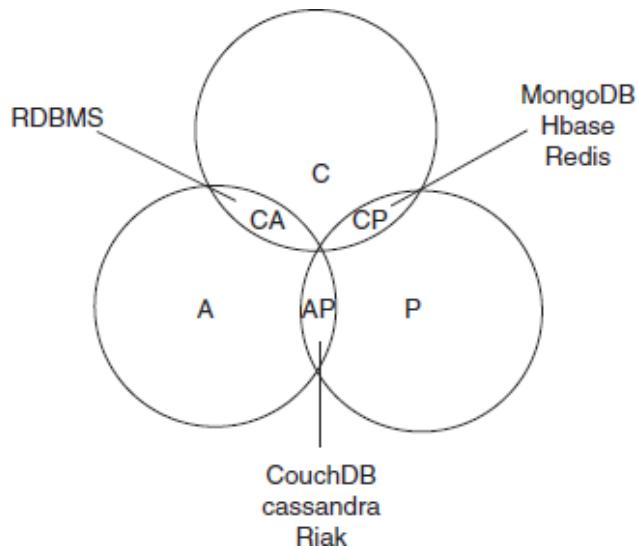
Looks like everything is in control. Wait a minute! There is a tiff that has taken place between the office admins. The two are pretty much available but are not talking to each other which, in other words, means that the updates are not flowing from one to the other. ***We have to be partition tolerant!!!*** As a training coordinator, you instruct them saying that none of you are taking any calls requesting for schedules or updates to schedules till you patch up. This implies that the system is partition tolerant but not available at that time.

In summary, one can at most decide to go with two of the three. Refer [Figure 1.15](#).

1. **Consistent:** The instructors or the training coordinator once they have updated information with the office admins, will always get the most updated information when they call up the office admins.
2. **Availability:** The instructors or the training coordinator will always get the schedule if any or both of the office admins have reported to work.
3. **Partition tolerance:** Work will go on as usual even if there is communication loss between the office admins owing to a spat or a tiff!

### ***When to choose consistency over availability and vice-versa***

1. Choose availability over consistency when your business requirements allow some flexibility around when the data in the system synchronizes.
2. Choose consistency over availability when your business requirements demand atomic reads and writes.



**Figure 1.15** Two of the three properties are adhered to in any NoSQL as per CAP theorem.

### TRY THIS – 3

### ACROSS

3. CAP theorem is also called as \_\_\_\_\_ theorem.
  4. System will continue to function even when network partition occurs.

## DOWN

1. Every read fetches the most recent write.
  2. A non-failing node will return a reasonable response within a reasonable amount of time.

## TRY THIS – 4

---

State databases that follow one of the possible three combinations:

1. Availability and Partition Tolerance (AP)
2. Consistency and Partition Tolerance (CP)
3. Consistency and Availability (CA)

## REMEMBER ME

---



1. Scalability = Ability of the system to handle growing amount of work.
2. Vertical scalability or scale up = Handle more load by using faster processors.
3. Horizontal scalability or scale out = Handle more load by using more processors.
4. NoSQL is a good choice for companies experiencing rapid growth with no clear schema definitions.
5. NoSQL handles (manage and store) unstructured, messy, and unpredictable data.
6. NoSQL databases are non-relational, open-source, and distributed databases.

Eric Brewer's CAP theorem states that in a distributed computing environment (a collection of interconnected nodes that share data), it is impossible to provide the following guarantees. At best you can have two of the following three, while one must be sacrificed.

1. Consistency.
2. Availability.
3. Partition tolerance.

## QUESTION ME

---

1. Explain the ACID properties of transaction.
2. What is your interpretation of “Basically Available Soft State Eventual Consistency”?
3. Explain your understanding of Eric Brewer's “CAP” theorem.
4. What is a distributed system?
5. Will NoSQL databases replace RDBMS?

## 6. How does NoSQL compare with regards SQL?

### REFERENCE ME

---

1. <https://www.mongodb.com/nosql-explained>
2. <http://nosql-database.org/>
3. <https://www.geeksforgeeks.org/introduction-to-nosql/>
4. <https://en.wikipedia.org/wiki/NoSQL>

### ANSWERS

---

#### Try This – 1

(a) and (d).

#### Try This – 2

See the table below for reference.

SQL	NoSQL
Relational	Distributed
Predefined schema	Wide-column stores
Vertically scalable	Key-value pairs
MySQL	CouchDB
Acid properties	Neo4j
Table or relations	Cassandra
	Large dataset
	Brewers CAP theorem
	Document based database
	Scales horizontally
	Avoids join operations
	JSON data

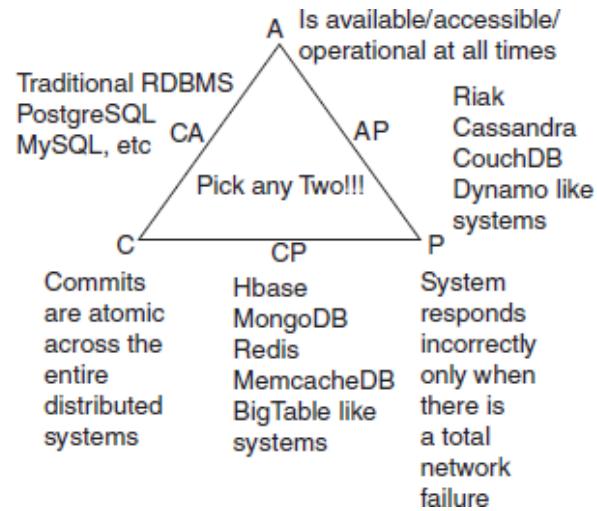
## Try This – 3

## Try This – 4

Examples of databases that follow one of the possible three combinations:

1. Availability and Partition Tolerance (AP)
    - (a) Riak
    - (b) Cassandra
    - (c) CouchDB
    - (d) Dynamo
  2. Consistency and Partition Tolerance (CP)
    - (a) HBase
    - (b) MongoDB
    - (c) Redis
    - (d) Memcached
    - (e) BigTable
  3. Consistency and Availability (CA)
    - (a) Traditional RDBMS such as PostgreSQL, MySQL, etc.

Refer [Figure 1.16](#) to understand this better.



**Figure 1.16** CAP Theorem.

*OceanofPDF.com*



# Types of NoSQL Databases

---

## BRIEF CONTENTS

- Introduction
- Key–Value Pair Databases
  - What are Keys?
  - What are Values?
  - Differences between Key–Value and Relational Databases
- Document Databases
  - Querying Documents
  - Differences between Document and Relational Databases
- Column-Family Databases
  - Comparison of Column-Family Database with Key–Value and Document
- Graph Database
  - Understanding Graph Database

---

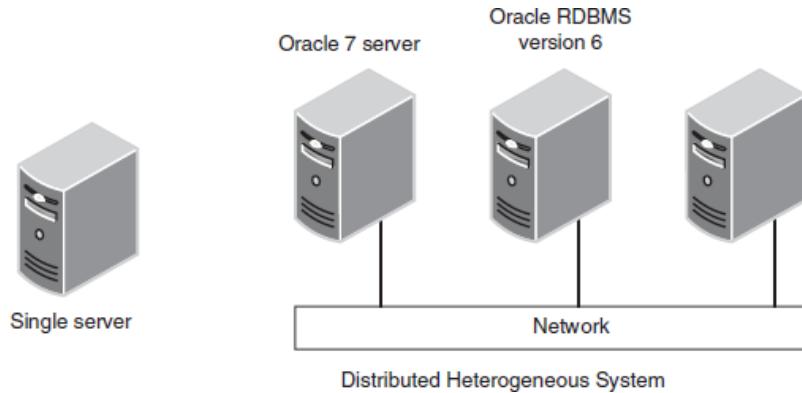
## 2.1 INTRODUCTION

Usually we hear about NoSQL databases as distributed databases (a distributed database is stored across multiple computers which are either located in the same physical location or dispersed over a network of interconnected computers). The question here is: “Are NoSQL databases always required to be implemented as distributed systems?” The answer is “No”. Most of them can run on a single server. However, let us be cognizant of the fact that majority of the likeable and remarkable features can be witnessed when the NoSQL database is setup as a distributed system (setup spread across multiple servers). Distributed environments often times lead to concerns about consistency, availability, partition tolerance, scalability, etc. Refer [Figure 2.1](#).

There are four types of NoSQL databases:

1. Key-value pair databases.
2. Document databases.
3. Column family store databases.
4. Graph databases.

In the following sections we will discuss about these various types of NoSQL databases.



**Figure 2.1** Single server versus distributed multi-server system.

## 2.2 KEY-VALUE PAIR DATABASES

Key-value databases are the simplest of all the NoSQL databases. They are based on a very simple model where we have keys (unique identifiers) to look up the data.

### 2.2.1 What are Keys?

Keys are unique identifiers. As the term identifier suggests, they are used to uniquely identify (pickup) a value from a set of values.

Let us look at a simple example of a phone directory (Figure 2.2).

Phone directory	
Key	Value
Kane	(123) 456-7890
Jannet	(234) 567-8901
Kiara	(345) 678-9012
Turing	(456) 789-0123

**Figure 2.2** Sample key-value store.

Given the key, the value can be very easily fetched. If you provide the value as “Kiara”, the value of (345) 678-9012 will be returned.

Let us look at another example where we are required to capture the below information about each book available in the library.

1. Book Name.
2. Author Name.
3. Date of Publication.

#### 4. Book Type.

Let us store this information in a key–value format. For each book, let us create a unique key. The simplest process will be to use a sequential number for each book and then append the name of the attribute to it. For example, data about the first book in the system will use keys 1.BookName, 1.AuthorName, 1.YearofPublication, 1.BookType (Figure 2.3).

Key	Value
1.BookName	Data Analysis Using R
1.AuthorName	Seema Acharya
1.YearofPublication	2018
1.BookType	Paperback

Figure 2.3 Sample key–value store to store information about the book.

The important point to note here is that the key has to be unique. Where are the keys stored in the database? They are stored in the namespace/keyspace. What is a namespace in database terminology? A namespace is a collection of identifiers. The keys must be unique within a namespace. The database can have one namespace or multiple namespaces. A namespace could correspond to an entire database. In such a case all the keys in the database should be unique.

Examples of a database with single namespace and multiple namespaces are shown in Figure 2.4 and Figure 2.5, respectively.

Key	Value
1.BookName	Data Analysis Using R
1.AuthorName	Seema Acharya
1.YearofPublication	2018
1.BookType	Paperback

Figure 2.4 Database with a single namespace.

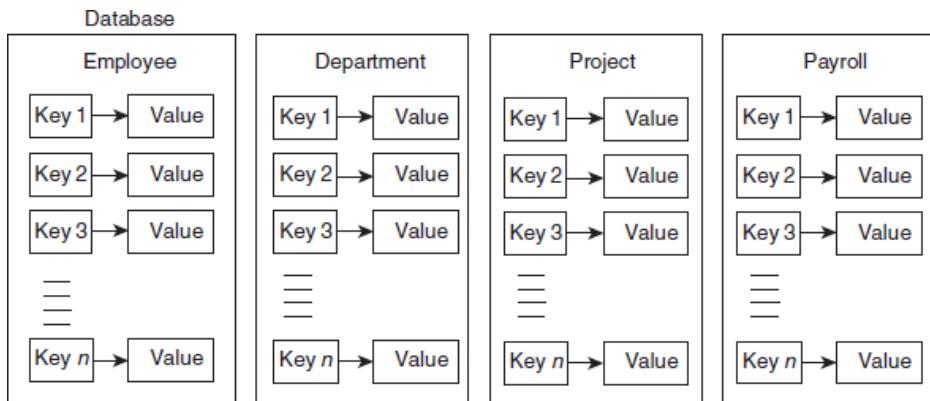
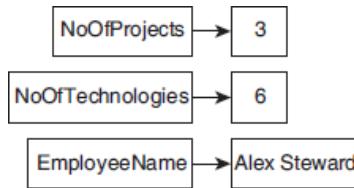


Figure 2.5 Database with multiple namespaces to store information about “Employee”, “Department”, “Project”, and “Payroll”.

#### 2.2.2 What are Values?

Values are data stored along with keys.

*What data type can be used to store values?* It can be as simple as a number such as the number of projects that the employee in an organization has worked on, the number of technologies he/she is familiar with, etc. It can be a string value (long or short string) such as “name of the employee”, etc. (refer [Figure 2.6](#)). In a key-value pair database, one can store complex values as well such as images or binary objects. An example where image is used in the value could be to identify employees in the organization. The key could be “EmployeeID.photo” and the value could be the picture/photo stored as a binary large object type (BLOB). Typically, the key-value databases do not impose any checks on the data types of values. It is left to the programmer to determine the range of valid values and enforce the same.



**Figure 2.6** Sample key-value pairs.

### 2.2.3 Differences between Key–Value and Relational Databases

1. Key–value databases are simple, non-trivial data models. Relational databases, on the other hand, have strict adherence to relational data model.
2. **Data of Indeterminate Form:** RDBMS (Relational Database Management System) has strict adherence to schema. Before pushing the data into RDBMS relations, it is mandatory to define a schema. This makes it difficult to store HTML pages within a relation. This is primarily because each HTML page is different. Defining a schema for such page is complex. Key–value data store does not require a schema and it would be a best fit for such data.
3. **Data of Huge Size and Quantity:** RDBMS stores data in relations. Relations comprise rows and columns. They are optimized for small rows that support table fitting within a single server. On the contrary, key–value data stores are typically implemented as distributed systems that support storing large objects with huge quantity spread across multiple servers.
4. **Unrelated Data:** As you already know, logically related data is stored in database but what do you do if the requirement is to store unrelated data. In key–value pair data stores, there are no tables and therefore no features of tables/relations such as columns, constraints such as Not Null, Check constraint, etc. There is no need of joins in key–value databases as there are no foreign keys. Therefore key–value pair databases are the ideal choice for unrelated data.
5. Key–value databases do not support a rich query language such as SQL. Relational databases make querying the database easy owing to the great support provided by SQL.

There is, however, some similarity in the key naming convention.

Assume a table “Employee” with attributes (EmpID (primary key), EName, DeptName, ProjectName ....). The key in a key–value store can be Employee.EmpID.ColumnName. Here “Employee” is the name of the table, “EmpID” is the primary key, and “ColumnName” is the name of the column that you wish to refer to. Refer [Figure 2.7](#) which shows the naming convention in this key–value database.

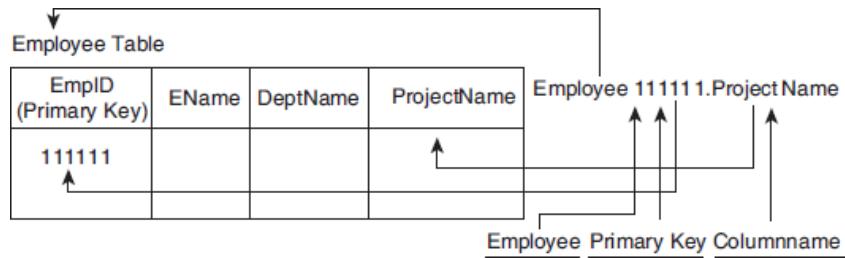


Figure 2.7 Naming convention in a key-value database.

## 2.3 DOCUMENT DATABASES

Document databases are also called *document-oriented databases*. There is some similarity with key-value stores as they too are a collection of key values. The difference lies in what is stored as values. Documents are stored as values. These documents are semi-structured entities such as JSON (Java Script Object Notation) or XML (Extensible Markup Language).

**Example:** A document to store information about a customer.

```
{
  CustomerID: 111111,
  FirstName: "ABC",
  LastName: "XYZ",
  Location: "Canada",
  ContactNo: 333-222-1111,
  EmailID: ABC_XYZ@zzz.com,
}
```

As can be seen from the example document, the document comprises several attributes: *CustomerID*, *FirstName*, *LastName*, *Location*, *ContactNo*, *EmailID*, etc.

A couple of quick questions here...

1. *Is it required to define a schema prior to placing documents in a document database?*

Unlike RDBMS, a document database does not need a schema to be defined prior to placing data in a document. When a document is added to a database, it creates the underlying data structures needed to support the document.

2. *Is it required for all the documents in a collection (collection is analogous to a table in RDBMS) to have the same set of attributes?*

The answer is “No”.

We can have another document as below as part of the same collection:

```
{
  CustomerID: 222222,
  FirstName: "DEF",
  LastName: "JKL",
  Location: "Canada",
  Gender: "Male"
  ContactNo: 333-222-1111,
  EmailID: ABC_XYZ@zzz.com,
}
```

### 2.3.1 Querying Documents

In key-value databases, the only way to look up a value is using the key. However, document databases provide you the flexibility to look up a document based on attribute values.

**Example:**

```
db.Customer.find({Location: "Canada"});
```

The above statement will return all documents with the location “Canada”. In the command, “Customer” is the name of the collection and “Location” is the name of the column.

As with relational databases, document databases typically support operators such as AND, OR, greater than, less than, and equal to.

### 2.3.2 Differences between Document and Relational Databases

Let us begin with terminology used in RDBMS with its equivalent in MongoDB, a document database ([Table 2.1](#)).

**Table 2.1** Terminology used in RDBMS and its equivalent in MongoDB

RDBMS	MongoDB
Database	Database
Table	Collection
Record	Document
Columns/fields/attributes	Key-value pairs
Primary key	Unique Identifier

Relational databases (RDBMS) have strict adherence to relational data model. They require a fixed, pre-defined schema. Each record has the same set of columns. A document database, on the other hand, does not have a fixed schema. Each document can have its own set of attributes/key-value pairs.

As in RDBMS, in a document store too, it is possible to search on multiple attributes.

To bring out the difference clearly, let us picture how employee details are stored in RDBMS and how they are stored in a document database such as MongoDB. Consider the following tables: “Employee”, “EmployeeSkill”, and “EmployeeProjects” in RDBMS.

#### Employee Table

EmployeeID	EmployeeFName	EmployeeLName	Designation	Department	DateofJoining
E1001	Sam	Mathew	Project Manager	Information Systems	4 Feb. 2004
—	—	—	—	—	—

EmployeeSkill Table

EmployeeID	TechSkill	SkillLevel
E1001	Microsoft PowerBI	Advanced
E1001	Microsoft Azure PowerBI	Beginner
E1001	Microsoft Azure Data Warehouse	Advanced
—	—	—

EmployeeProjects Table

EmployeeID	ProjectID	ProjectName	ProjectStartDate	ProjectEndDate	Technology
E1001	P1001	Project AirWings	01-March-2007	31-March-2009	Microsoft
E1001	P1010	Project Eagle	12-April-2010	30-May-2014	Business Intelligence

As you can see from the above structure, if we need to get the details about employees along with the technologies that they are skilled in and also the details about the projects that they have been involved with, it will require a join between the three tables. Joins are an overhead and may turn out to be a performance bottleneck.

The same information can be stored in a document database like MongoDB. This is shown below:

```

{
EmployeeID: "E1001",
EmployeeFName: "Sam",
EmployeeLName: "Mathew",
Designation: "Project Manager",
Department: "Information Systems",
DateofJoining: "4-Feb-2004",
EmployeeSkill:
[
{
TechSkill: "Microsoft PowerBI",
SkillLevel: "Advanced"
}
{
TechSkill: "Microsoft Azure Power BI",
SkillLevel: "Beginner"
}
{
TechSkill: "Microsoft Azure Data warehouse",
SkillLevel: "Advanced"
}
]
EmployeeProjects:
[
{
ProjectID: "P1001",
ProjectName: "Project AirWings",
ProjectStartDate: "01-Mar-2007",
ProjectEndDate: "31-Mar-2009",
Technology: "Microsoft"
}
{
ProjectID: "P1010",
ProjectName: "Project Eagle",
ProjectStartDate: "12-Apr-2010",
ProjectEndDate: "30-May-2014",
Technology: "Business Intelligence"
}
]
}

```

Embedding documents or lists of values in a document eliminates the need for joining documents. This saves much overhead.

### TRY THIS – 1

- 
1. NoSQL databases were designed with security in mind, this implies that developers or security teams do not need to worry about implementing a security layer. State whether this statement is True or False. Give reasons for your answer.
  2. NoSQL databases have become a popular solution for quite a few organizations. Which of the following is not a reason for its immense popularity?

- (a) Better scalability.  
 (b) Improved ability to keep data consistent.  
 (c) Faster access to data than relational database management systems.  
 (d) Allows for data to be held across multiple servers more easily.
3. NoSQL database prohibits Structured Query Language (SQL). State whether this statement is True or False. Give reasons for your answer.
4. When is it best to use a NoSQL database?
- (a) When confidentiality, integrity, and availability of data is paramount.  
 (b) When the data is predictable.  
 (c) When there is a need to retrieve huge quantities of data.  
 (d) When the speed of retrieval of data is not critical.

## 2.4 COLUMN-FAMILY DATABASES

Let us think back for a moment on “big data”. How much data is big data? Is it millions of records? Well, may be or may be not! Big data is a subjective term. To some the millions of records in any open-source database such as MySQL or PostgreSQL is big, to some it is average, and to a few others, it is small. How about “billions of records” with “tens of thousands of columns” in a table? This is by all means a very large database.

Let us look at how RDBMS, key-value, and document databases store the billions of records and tens of thousands of columns in a table.

1. Can RDBMS store such volume? The answer is “Yes”. We have the “Very Large Databases (VLDBs)”, but they come with a significant cost.
2. Key-value data stores can also be scaled to support this volume. However, they have limitations with organizing data in columns, and also with storing and querying frequently used data together.
3. Document databases are our next consideration. Here though scale is not a concern, the lack of an RDBMS style SQL-like query language sure is.

Constraints such as the above prompted Google to come up with “BigTable” in 2006. Thus was born a new kind of database called *column-family database*.

Before we proceed ahead, let us understand few terminologies such as “row-oriented storage”, “column-oriented storage”, and “column families”:

1. **Row-Oriented Storage:** In row-oriented storage, data is organized into rows. A row comprises several columns.

**Example:** A row to store the details of an Employee.

EmpID, EmpName, Designation, DepartmentID, EmailID, Country, State, City, Zipcode

EmpID	EmpName	Designation	DepartmentID	EmailID	Country	State	City	ZipCode
-------	---------	-------------	--------------	---------	---------	-------	------	---------

2. **Column-Oriented Storage:** Here as the name suggests, data is organized into columns.

**Example:** A column “EmpID” to store the employee numbers of employees of an organization.

EmplD
E1001
E1002
E1003
E1004
E1005
E1006

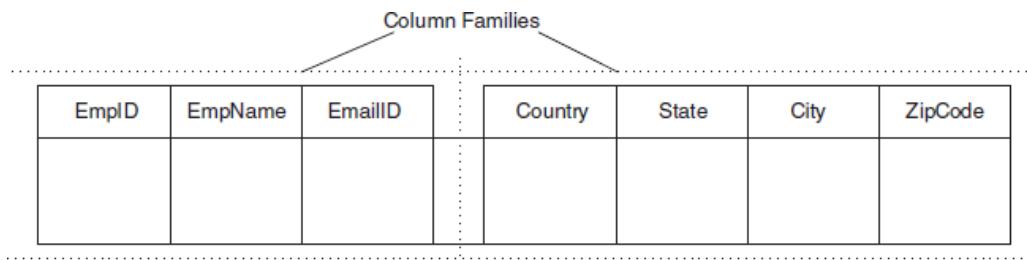
- 3. Column Families:** Column families are composed of columns that are frequently accessed together. Therefore, it makes sense to store/keep them together.

**Example:** The following are two column families

Column Family 1: EmplD, EmpName, EmailId, ...

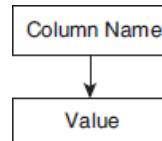
Column Family 2: Country, State, City, ZipCode, ...

Refer [Figure 2.8](#).

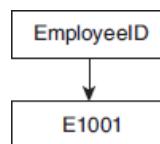


**Figure 2.8** Column families.

As the name suggests, in a column-family database there are columns. A column has a name and a value.



**Example:** A column by the name “EmployeeID” with a value “E1001”.



A set/group of columns makes up a row.

**Example:** A row comprising columns EmplD, EmpName, EmailID, etc.

EmplD	EmpName	EmailID
-------	---------	---------

Rows can be homogeneous (all rows having the same set of columns) or heterogeneous (rows can have different columns). Refer [Figure 2.9](#) for an example of homogeneous rows and [Figure 2.10](#) for heterogeneous rows.

UserID	UserName	Location
U101	AAA	New york
U102	BBB	Dallas

**Figure 2.9** Homogeneous rows.

UserID: U101	UserName: AAA	Location: New york		
UserID: U102	UserName: BBB	Location: Dallas	Contact No:.....	Alternate contact No:.....

**Figure 2.10** Heterogeneous rows.

We can make groups of related columns. These groups of related columns are called column families. A row can have many column families. Column families for a row may or may not be stored together however all the columns in a column family are always stored together. The columns which are frequently accessed together comprise the column family.

#### 2.4.1 Column-Family Database versus Key–Value and Document

**1. Key–Value Database:** Keys are the identifiers to look up values in a key–value database. They cannot be used if the requirement is to query by data, have relationships between the data being stored, or if we need to operate on multiple keys at the same time.

Refer [Figure 2.11](#). Here “Country”, “State”, and “City” are the keys and “United States of America”, “New York”, and “New York City” are the values.

Key–value databases are generally used to store

- (a) Session information
- (b) User profiles
- (c) Use preferences
- (d) Shopping cart data, etc.



**Figure 2.11** Key–value pairs in key–value database.

**2. Document Databases:** Here data is stored in documents. Refer [Figure 2.12](#). They are not preferred for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures. Document databases are generally useful for:

- (a) Content management systems
- (b) Blogging platforms
- (c) Web analytics

- (d) Real-time analytics
- (e) E-commerce applications, etc.

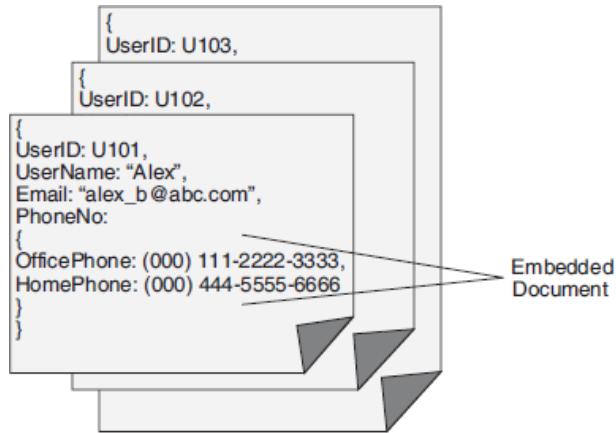


Figure 2.12 Sample documents in a document database.

- 3. Column-Family Databases:** In column-family databases, data is stored in rows. Each row has a unique row identifier. A row comprises several columns. Columns which are frequently used together are stored together in column families. Figure 2.13 shows two rows with row identifiers “Row Key 1” and “Row Key 2”. Both the rows are composed of columns: “Col A”, “Col B”, “Col C”, and “Col D”. There are two column families, where “Col A” and “Col B” constitute the first column family, and “Col C” and “Col D” constitute the second column family.

They are generally useful for

- (a) Content management systems
- (b) Blogging platforms
- (c) Maintaining counters
- (d) Expiring usage
- (e) Heavy write volume such as log aggregation

Avoid using column-family databases for systems that are in early development, have changing query patterns, etc.

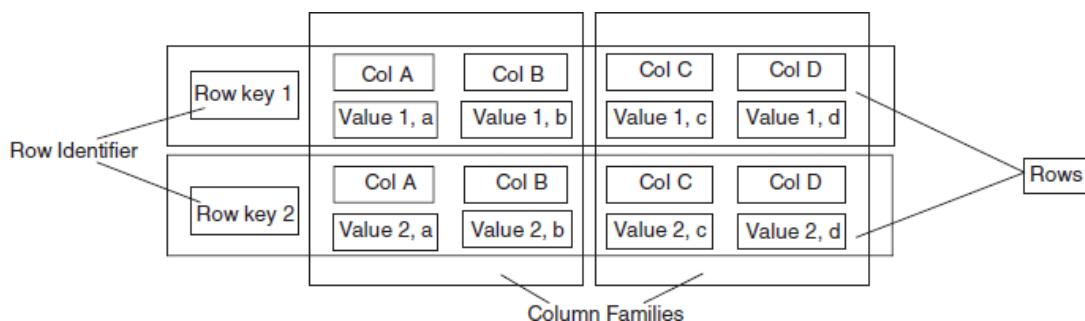


Figure 2.13 Sample column family database.

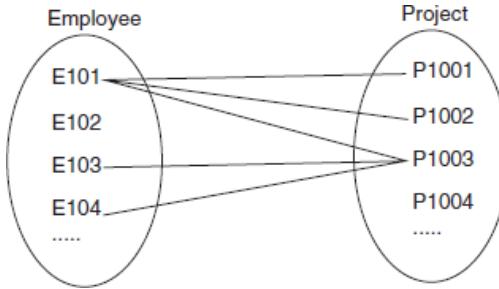
## 2.5 GRAPH DATABASE

## Picture This

An employee with Employee ID “E101” has been with an enterprise for close to 6 years. Currently he is working on three different projects the details of which are as follows:

1. Project P1001 – Technology used was Java. Working in the role of technology architect.
2. Project P1002 – Technology used was Microsoft. Working as a consultant on the project.
3. Project P1003 – Technology used was Java. Working as a tester.

An employee can work on one or more projects at any point in time. Also a project will have several employees working on it. It is a clear case of “Many-to-Many” relationship.



**Figure 2.14** Many-to-many relationship between “Employee” and “Project” entity sets.

Refer [Figure 2.14](#). Employee E101 works on projects P1001, P1002, and P1003. A project P1003 has employees E101, E103, and E104 working on it.

If we were to store this information into relational database, we will make use of three tables. An employee table to store details about the employees ([Table 2.2](#)), a project table that would carry details about the projects ([Table 2.3](#)), and a join table (also called as an associative entity table) that will store information on which employees worked in which projects ([Table 2.4](#)).

**Table 2.2** Employee table

EmployeeID	EmployeeName	DateOfJoining	Experience	BaseLocation	CurrentLocation	—
E101	Akshay Sharma	12-04-2013	06 years	Pune	Pune	—
—	—	—	—	—	—	—

**Table 2.3** Project table

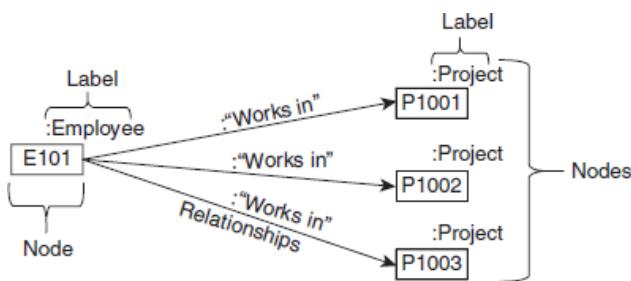
ProjectID	ProjectName	ProjectManager	ClientName	ProjectStartDate	—
P1001	AAA	Alex	XXX	12-May-2013	—
P1002	BBB	Bob	YYY	11-May-2015	—
P1003	CCC	Andrews	ZZZ	22-Mar-2016	—

**Table 2.4** Employee\_Project table

EmpID	ProjectID
E101	P1001
E101	P1002
E101	P1003

Now assume it is required to fetch the details of all the employees along with details of the projects that they are working on. This will mean that we perform two lookups. Let us pick up the first record from the “Employee\_Project” table. The EmpID is E101 and the ProjectID is P1001. Use the employee ID value of E101 to look up and fetch the details of the employee from [Table 2.2](#). Likewise use the project ID value of P1001 to look up and fetch the details of the project from [Table 2.3](#).

Now let us represent the same data in a graph database. A graph database has nodes and edges. Nodes represent entities and edges represent relationship between entities.



**Figure 2.15** Graph to represent relationship between “Employee” and “Project”.

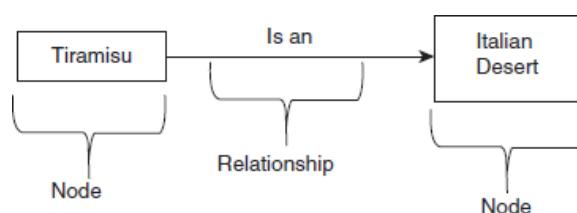
Refer [Figure 2.15](#). We have a single node for employee E101 with a label of “Employee”. Employee E101 works in three different projects, namely, P1001, P1002, and P1003. Therefore, we create a node for each project with a label of “Project”. To find out which projects, employee, “E101” is working on, we would search the graph for employee E101’s node, then traverse all of the “Works\_in” relationships from node E101 to find the project nodes he/she is connected to. That is all we need – a single hop with no lookups involved.

### 2.5.1 Understanding Graph Database

A graph comprises two elements:

- 1. A node:** A node represents an entity such as a “person”, “employee”, “customer”, “place”, “book”, “project” etc.
- 2. A relationship:** A relationship represents how two nodes are related/associated.

**Example:** Tiramisu is an Italian desert. Here “Tiramisu” and “Italian desert” will be represented by two nodes and “is an” will constitute the relationship between the two nodes. Refer [Figure 2.16](#).



**Figure 2.16** Sample nodes and the relationship between them.

The reasons behind the immense popularity of graph database today is

1. The real world is richly interconnected.
2. It maps more realistically to how the human brain maps and processes the world around it.
3. They mimic sometimes-consistent, sometimes-erratic relationships in an intuitive way. That is what makes the graph paradigm different than other database models.

### TRY THIS – 2

- 
1. Which of the following companies developed Apache Cassandra?
    - (a) LinkedIn
    - (b) Twitter
    - (c) MySpace
    - (d) Facebook
  2. Which of the following is used to store information about networks, such as social connection?
    - (a) Key-value
    - (b) Wide-column
    - (c) Document
    - (d) Graph
- 

### REMEMBER ME



- 
1. There are four types of NoSQL databases:
    - (a) Key-value pair databases
    - (b) Document databases
    - (c) Column family store databases
    - (d) Graph databases
  2. Key-value databases are the simplest of all the NoSQL databases. They are based on a very simple model where we have keys (unique identifiers) to look up the data.
  3. Document databases are also called *document-oriented databases*. There is similarity with key-value stores as they too are a collection of key values.
  4. Unlike RDBMS, a document database does not need a schema to be defined prior to placing data in a document.
  5. In column family databases, data is stored in rows. Each row has a unique row identifier. A row comprises several columns. Columns which are frequently used together are stored together in column families.
  6. The reasons behind the immense popularity of graph database today is:
    - (a) The real world is richly interconnected,
    - (b) It maps more realistically to how the human brain maps and processes the world around it.
    - (c) They mimic sometimes-consistent, sometimes-erratic relationships in an intuitive way. That is what makes the graph paradigm different than other database models.

## TEST ME

---

1. NoSQL databases is used mainly for handling large volumes of what type of data?
  - (a) Unstructured
  - (b) Structured
  - (c) Semi-structured
  - (d) All of the above
2. Most NoSQL databases support automatic \_\_\_\_\_. This implies that you get high availability and disaster recovery.
  - (a) Processing
  - (b) Scalability
  - (c) Replication
  - (d) All of the above
3. Which of the following is the simplest NoSQL database?
  - (a) Key-value
  - (b) Wide-column
  - (c) Document
  - (d) All of the above
4. Which of the following statements is incorrect?
  - (a) Non-relational databases require that schemas be defined before you can add data.
  - (b) NoSQL databases are built to allow the insertion of data without a predefined schema.
  - (c) NewSQL databases are built to allow the insertion of data without a predefined schema.
  - (d) All of the above
5. Which of the following is a wide-column store?
  - (a) Cassandra
  - (b) Riak
  - (c) MongoDB
  - (d) Redis
6. Which of the following is a NoSQL Database Type?
  - (a) SQL
  - (b) Document databases
  - (c) JSON
  - (d) All of the above

## QUESTION ME

---

1. How is data stored in a key-value database?
2. How is data stored in document database?
3. What is column-family data store?
4. Explain the difference between RDBMS, key-value, document, and column-family databases.
5. Is it mandatory for a NoSQL database to be distributed?
6. When should you avoid using a key-value database?
7. When should you avoid using a document database?
8. Name a few popular key-value, document, and column-family databases.

9. List three significant differences between relational databases and NoSQL databases.
10. Distributed environments often times lead to concerns about consistency, availability, partition tolerance, scalability, etc. Comment.

## REFERENCE ME

---

1. [https://en.wikipedia.org/wiki/Key-value\\_database](https://en.wikipedia.org/wiki/Key-value_database)
2. <https://aws.amazon.com/nosql/key-value/>
3. <https://aws.amazon.com/nosql/document/>
4. [https://en.wikipedia.org/wiki/Column-oriented\\_DBMS](https://en.wikipedia.org/wiki/Column-oriented_DBMS)
5. [https://en.wikipedia.org/wiki/Graph\\_database](https://en.wikipedia.org/wiki/Graph_database)
6. <https://neo4j.com/developer/graph-db-vs-rdbms/>

## ANSWERS

---

### Try This – 1

1. False.

**Reason:** Most of the popular NoSQL databases are open source. They were not designed with security as a priority.

2. (b) Improved ability to keep data consistent.

**Reason:** NoSQL Databases do not have strict adherence to Atomicity, Consistency, Isolation and Durability (ACID) properties of RDBMS. NoSQL databases are schema-less and permissions on a table, column or row cannot be segregated.

3. False

**Reason:** Contrary to the name, the approach to data management does support SQL elements.

4. (c) When there is a need to retrieve huge quantities of data

**Reason:** NoSQL is the best choice for huge volume and varieties of data (structured, semi-structured, and unstructured). RDBMS is an appropriate choice when consistency and availability of data is of paramount significance.

### Try This – 2

1. (d) Facebook.
2. (d) Graph database.

**Reason:** Graph databases are the best choice when it comes to depicting relationships between data.

## Test Me

1. (a) Unstructured

**Reason:** NoSQL databases can store huge volumes and varieties of data (structured, semi-structured and unstructured). It is basically used for housing huge volumes of unstructured

data. RDBMS is still a database of choice when it comes to structured data.

**2. (c) Replication**

**Reason:** Replication helps with high availability and disaster recovery. Replication is storing the same piece of data on two or more data nodes.

**3. (a) Key-value**

**Reason:** In key–value NoSQL databases, a key is used to identify a value or a set of values. This is by far the simplest NoSQL database. Few examples of key–value database are Amazon DB, Redis, and Riak.

**4. (a) Non-relational databases require that schemas be defined before you can add data.**

**Reason:** NoSQL databases are schema less and do not require a schema to be defined before dealing with data.

**5. (a) Cassandra**

**Reason:** Redis and Riak are key–value stores, while MongoDB is a document database.

**6. (b) Document databases**

**Reason:** SQL refers to RDBMS. JSON is Java Script Object Notation. MongoDB stores data as JSON documents.



# Column-Family Store

---

## BRIEF CONTENTS

- Introduction to Apache Cassandra
  - Cassandra versus RDBMS
- Features of Cassandra
  - Peer to Peer
  - Gossip and Failure Detection
  - Partitioner
  - Replication Factor
  - Writes in Cassandra
  - Hinted Handoff
  - Tunable Consistency
- Cassandra Query Language Data Types
- Cassandra Query Language Shell (CQLSH)
  - Keyspaces
- Collections
  - More Practice on Collections (SET and LIST)
- Cassandra Counter Column
- Time-to-Live (TTL)

- Alter Commands
- Import from and Export to CSV
  - Export to CSV
  - Import from CSV
  - Import from STDIN
  - Export to STDOUT
- Querying System Tables

### 3.1 INTRODUCTION TO APACHE CASSANDRA

---

Apache Cassandra is one of the most popular, open-source, distributed, wide column database management system. It is an immensely popular NoSQL owing to its high availability, linear scalability, proven fault tolerance on commodity hardware as well as cloud infrastructure and good performance.

Apache Cassandra was born at Facebook. After Facebook open sourced the code in 2008, Cassandra became an Apache Incubator project in 2009 and subsequently became a top-level Apache project in 2010. It is built on Amazon's dynamo and Google's BigTable.

Cassandra does NOT compromise on availability. Since it does not have a master-slave architecture, there is no question of single point of failure. There are no network bottlenecks. Every node in the cluster is identical. This proves beneficial for business-critical applications that need to be always up and running and cannot afford to go down ever.

Some important points to remember about Cassandra are as follows:

1. It is a highly scalable (it scales out), high performance distributed database. It distributes and manages gigantic amount of data across commodity servers.
2. It is a column-family store database designed to support peer-to-peer symmetric nodes instead of the master-slave architecture.
3. It has adherence to the Availability and Partition Tolerance properties of CAP theorem. It takes care of consistency using BASE (Basically Available Soft State Eventual Consistency) approach.
4. It has support for multiple data centers across geographies.
5. It supports very quick reads and writes.
6. It provides a rapidly scalable and flexible storage infrastructure.

Few companies that have successfully deployed Cassandra and have benefitted immensely from it are:

1. Apple: They have 75000 nodes cluster storing over 10 PB of data.
2. Twitter: Twitter uses MySQL and Cassandra for data storage. They are still experimenting with using Cassandra.
3. Netflix: They have 2500 nodes cluster storing over 420 TB of data and servicing over one trillion requests per day.
4. Cisco
5. Adobe: Adobe uses Cassandra in Adobe® AudienceManager, a Digital Marketing Suite (DMS) product that consolidates, activates, and optimizes digitally addressable data from all sources.
6. eBay: It has over 100 nodes cluster housing roughly 250 TB of data.
7. Rackspace.

## Cassandra versus RDBMS

NoSQL and RDBMS's are designed to support different application requirements and typically they co-exist in most enterprises. [Table 3.1](#) illustrates how RDBMS compares with Cassandra.

**Table 3.1** RDBMS versus Cassandra

Characteristic	RDBMS	Cassandra
Database model	Relational	Non-relational
Supported datatype	Structured data	Both structured and unstructured data
Schema	Rigid	Flexible
Scalability	Vertical	Horizontal
Reliability	Single point of failure due to single node model	No single point of failure due to its masterless architecture
Supported applications	Centralized applications (e.g., ERP)	Decentralized applications (e.g., Web, mobile, IoT, etc.)
Data velocity	Moderate velocity data	High velocity data (devices, sensors, etc.)
Availability	Moderate to high availability	Continuous availability; no downtime
Data sources	Data coming in from one/few locations	Data coming in from many locations
Data volume	Usually maintains moderate data volume with purge	Maintains high data volume; retains forever

## TRY THIS – 1

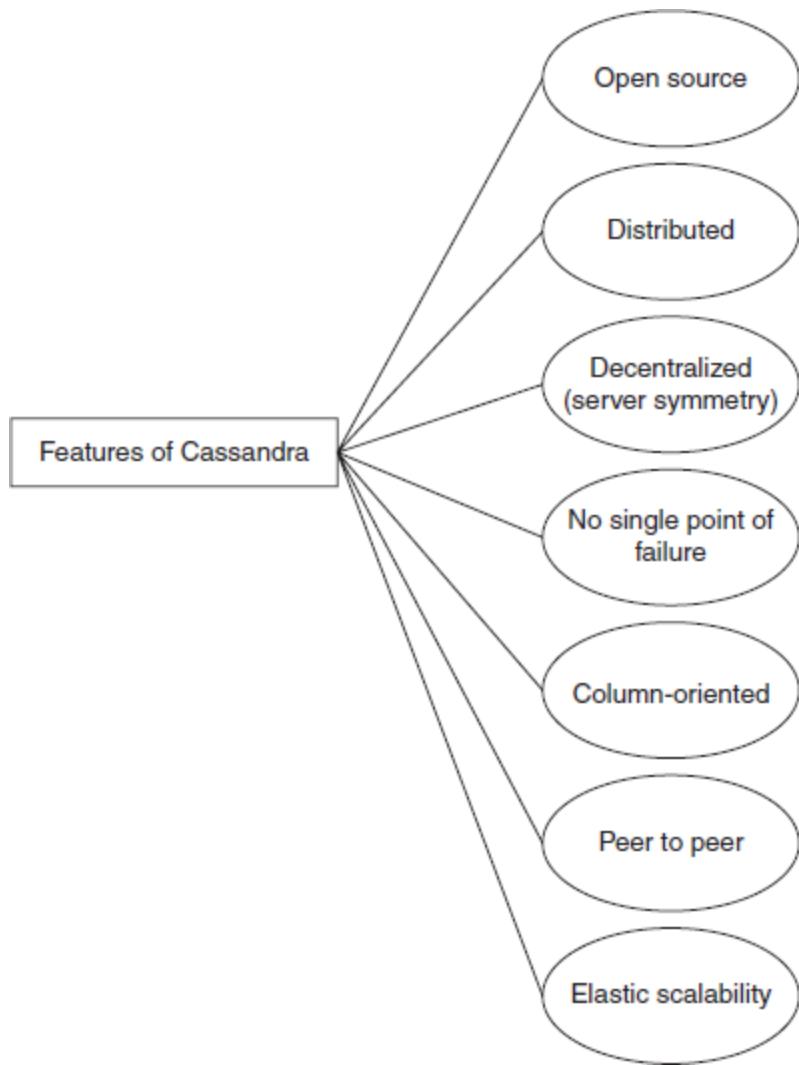
---

1. Cassandra is a NoSQL database of the following type:
    - (a) Key-value store
    - (b) Column-family store
    - (c) Document-oriented store
    - (d) Graph-based store
  2. An e-commerce website uses Cassandra to track the day-to-day activities of its customers. Which of the below prompted the selection of Cassandra as the backend store?
    - (a) Read heavy
    - (b) Write heavy
    - (c) Highly scalable
    - (d) Consistency
- 

## 3.2 FEATURES OF CASSANDRA

---

The features of Cassandra can be seen at a glance in [Figure 3.1](#).

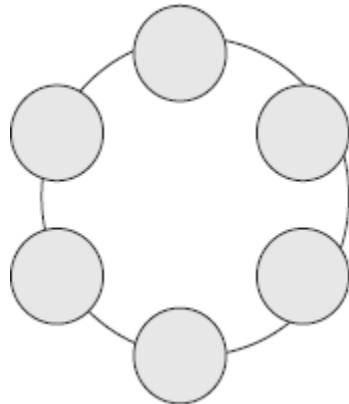


**Figure 3.1** Features of Cassandra.

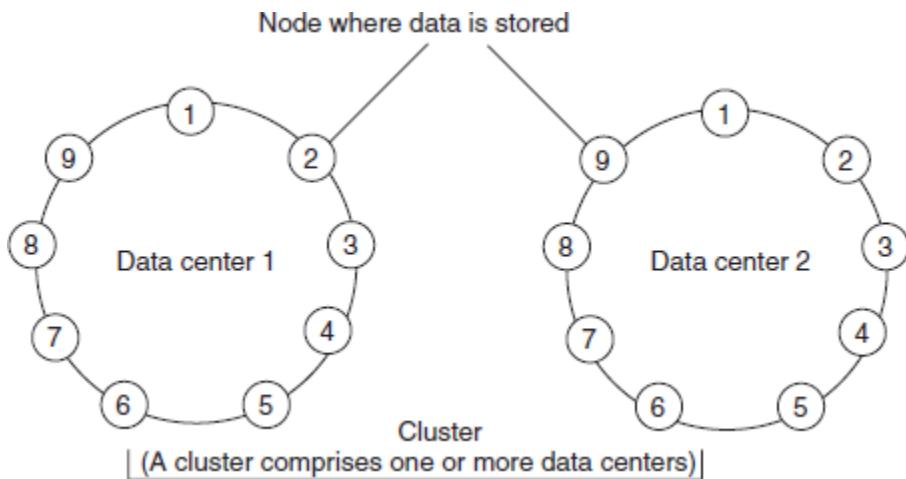
### 3.2.1 Peer to Peer

As with any other NoSQL database, Cassandra is designed to distribute and manage large data loads across multiple nodes in a cluster constituted of commodity hardware. Cassandra does NOT have a master-slave architecture, which means it does NOT have a single point of failure. A node in Cassandra is structurally identical to any other node. In case a node fails or is taken offline, it definitely impacts the throughput. However, it is a case of graceful degradation where everything does not come crashing at any given instant owing to a node failure. One can still go about business as usual. It tides over the problem of failure by employing a peer-to-peer distributed system across homogeneous nodes. It ensures that the data is distributed across all nodes in

the cluster. Each node exchanges information across the cluster every second. To summarize, Cassandra employs a peer-to-peer master-less architecture ([Figure 3.2](#)). All the nodes in the cluster are equal. Data is replicated on multiple nodes to ensure fault tolerance and high availability ([Figures 3.3](#) and [3.4](#)).



**Figure 3.2** A data center with logically related nodes.



**Figure 3.3** A cluster with multiple data centers.

Let us look at how a Cassandra node writes. Each write is written to the commit log sequentially. A write is taken to be successful only if it is written to the commit log. Data is then indexed and pushed to an in-memory structure called the *memtable*. When the in-memory data structure – the memtable – is full, the contents are flushed to SSTable (Sorted String) data file on the disk. The SSTable is immutable and is append-only. It is stored sequentially on the

disk and is maintained for each Cassandra table. Partitioning and replication of all writes are performed automatically across the cluster.

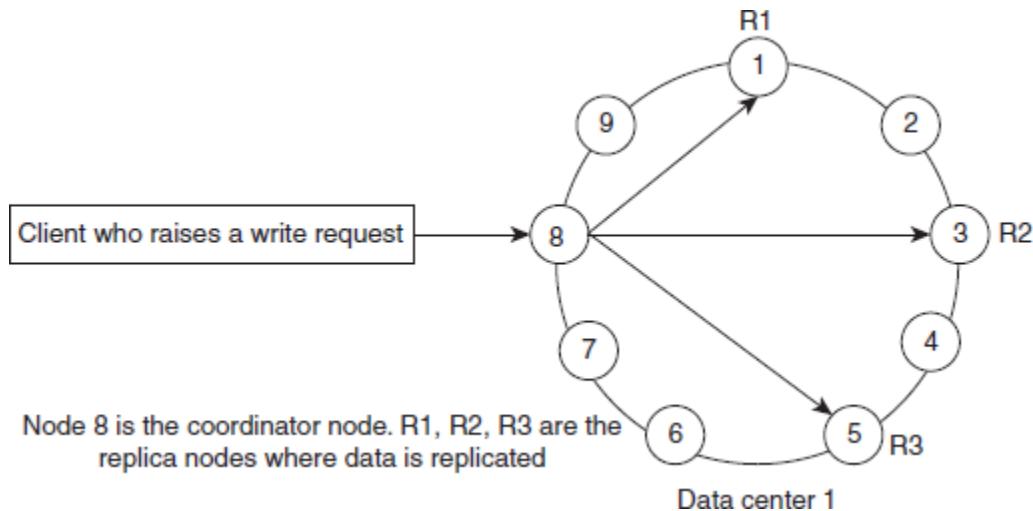


Figure 3.4 Coordinator and replication nodes.

### 3.2.2 Gossip and Failure Detection

Gossip protocol (Figure 3.5) is used for intra-ring communication. It is a peer-to-peer communication protocol which eases the discovery and sharing of location and state information with other nodes in the cluster.

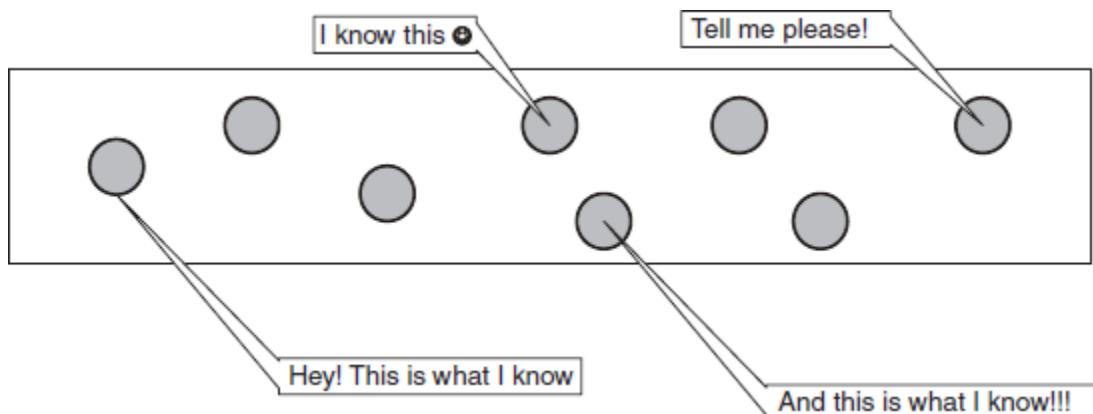


Figure 3.5 Gossip protocol.

### 3.2.3 Partitioner

A partitioner is a hash function to compute the token of the partition key. The partition key helps to identify a row uniquely. A partitioner takes a call on how to distribute data on the various nodes in a cluster. It also determines the node on which to place the very first copy of the data.

### **3.2.4 Replication Factor**

The replication factor determines the number of copies of data (replicas) that will be stored across nodes in a cluster. If one wishes to store only one copy of each row on one node, they should set the replication factor to 1. However if the need is for two copies of each row of data on two different nodes, one should go with a replication factor of 2. The replication factor should ideally be more than 1 and not more than the number of nodes in the cluster. A replication strategy is employed to determine on which nodes to place the data. Two replication strategies are available: (a) SimpleStrategy and (b) NetworkTopologyStrategy. The preferred one is NetworkTopologyStrategy because it is simple and supports easy expansion to multiple data centers, should there be a need.

### ***Anti-Entropy and Read Repair***

A cluster is made up of several nodes. Since the cluster is constituted of commodity hardware, it is prone to failure. In order to achieve fault-tolerance, a given piece of data is replicated on one or more nodes. A client can connect to any node in the cluster to read data. The number of nodes that will be read before responding to the client is based on the consistency level specified by the client. If the client-specified consistency is not met, the read operation blocks. There is a possibility that few of the nodes may respond with an out-of-date value. In such a case, Cassandra will initiate a read repair operation to bring the replicas with stale values up to date. The read repair operation is performed either before or after returning the value to the client as per the specified consistency level.

### **3.2.5 Writes in Cassandra**

Let us look at behind the scene activities. Here is a client who initiates a write request. Where does his/her write get written to? It is first written to the commit log. A write is taken as successful only if it is written to the commit

log. The next step is to push the write to a memory resident data structure called memtable. A threshold value is defined in the memtable. When the number of objects stored in the memtable reaches a threshold, the contents of memtable are flushed to the disk in a file called SSTable (Stored String Table). Flushing is a non-blocking operation. It is possible to have multiple memtables for a single column family. One out of them is current and the rest are waiting to be flushed.

### 3.2.6 Hinted Handoff

Cassandra is all for availability. It works on the philosophy that it will always be available for writes.

Refer [Figure 3.6](#). Assume that we have a cluster of three nodes: Node A, Node B and Node C. Node C is down for some reason. We are maintaining a replication factor of 2 which implies that two copies of each row will be stored on two different nodes. The client makes a write request to Node A. Node A is the coordinator and serves as a proxy between the client and the nodes on which the replica is to be placed. The client writes Row K to Node A. Node A then writes Row K to Node B and stores a hint for Node C. The hint will have the following information:

1. Location of the node on which the replica is to be placed.
2. Version metadata.
3. The actual data.

When Node C recovers and is back to the functional self, Node A reacts to the hint by forwarding the data to Node C.

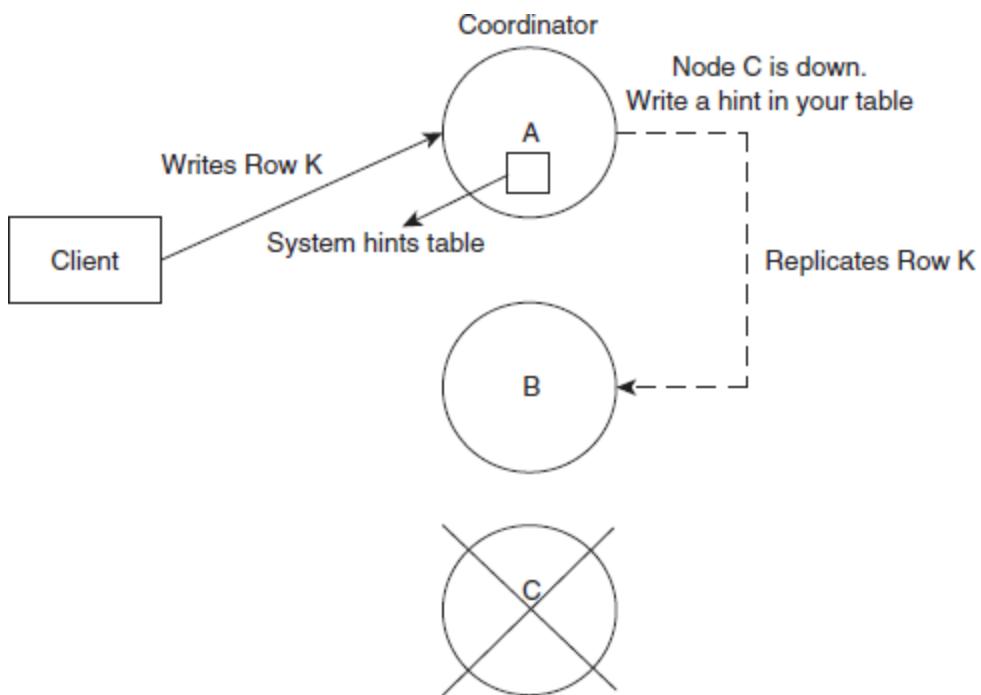
### 3.2.7 Tunable Consistency

#### Read Consistency

---

ONE	Returns a response from the closest node (replica) holding the data.
QUORUM	Returns a result from a quorum of servers with the most recent timestamp for the data.

LOCAL_QUORUM	Returns a result from a quorum of servers with the most recent timestamp for the data in the same data center as the coordinator node.
EACH_QUORUM	Returns a result from a quorum of servers with the most recent timestamp in all data centers.
ALL	This provides the highest level of consistency of all levels and the lowest level of availability of all levels. It responds to a read request from a client after all the replica nodes have responded.



**Figure 3.6** Hinted handoffs.

## Write Consistency

ALL	This is the highest level of consistency of all levels as it necessitates that a write must be written to the commit log and memtable on all replica nodes in the cluster.
-----	--

EACH_QUORUM	A write must be written to the commit log and memtable on a quorum of replica nodes in <i>all</i> data centers.
QUORUM	A write must be written to the commit log and memtable on a quorum of replica nodes.
LOCAL_QUORUM	A write must be written to the commit log and memtable on a quorum of replica nodes in the same data center as the coordinator node. This is to avoid latency of inter-data center communication.
ONE	A write must be written to the commit log and memtable of at least one replica node.
TWO	A write must be written to the commit log and memtable of at least two replica nodes.
THREE	A write must be written to the commit log and memtable of at least three replica nodes.
LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local data center.

---

### 3.3 CASSANDRA QUERY LANGUAGE DATA TYPES

The Cassandra Query Language (CQL) provides a rich set of built-in data types for columns. The counter type is unique. Besides these built-in data types, users can also create their own custom data types.

**Note:** Enclose ASCII text, timestamp values in single quotation marks. Enclose names of a keyspace, table, or column in double quotation marks.

[Table 3.2](#) provides a list of built-in data types in CQL.

**Table 3.2** CQL built-in data types

CQL data type	Description
Int	32-bit signed integer
Bigint	64-bit signed long
Double	64-bit IEEE-754 floating point
Float	32-bit IEEE-754 floating point
Boolean	True or false
Blob	Arbitrary bytes, expressed in hexadecimal
Counter	Distributed counter value
Decimal	Variable – precision integer
List	A collection of one or more ordered elements
Map	A JSON style array of elements
Set	A collection of one or more elements
Timestamp	Date plus time
Varchar	UTF 8 encoded string
Varint	Arbitrary-precision integers
Text	UTF 8 encoded string

## 3.4 CASSANDRA QUERY LANGUAGE SHELL (cqlsh)

Users communicate with the Cassandra database using CQL. Cassandra commands are executed at the CQL shell **cqlsh**. In other words, cqlsh is a command line shell for interacting with Cassandra through CQL. The syntax of CQL is similar to SQL.

### *Logging into cqlsh*

```
Command Prompt - cqlsh
Microsoft Windows [Version 10.0.16299.726]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\seema_acharya>cqlsh

WARNING: console codepage must be set to cp65001 to support utf-8 encoding on Windows platforms.
If you experience encoding problems, change your console codepage with 'chcp 65001' before starting cqlsh.

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.3 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
WARNING: pyreadline dependency missing. Install to enable tab completion.
cqlsh>
```

### *Objective:*

To get help with CQL.

*Act:*

```
help
```

*Outcome:*

```
Command Prompt - cqlsh
cqlsh> help

Documented shell commands:
=====
CAPTURE  CLS          COPY  DESCRIBE  EXPAND  LOGIN   SERIAL  SOURCE  UNICODE
CLEAR    CONSISTENCY  DESC   EXIT       HELP    PAGING  SHOW    TRACING

CQL help topics:
=====
AGGREGATES          CREATE_KEYSPACE      DROP_TRIGGER    TEXT
ALTER_KEYSPACE       CREATE_MATERIALIZED_VIEW  DROP_TYPE     TIME
ALTER_MATERIALIZED_VIEW CREATE_ROLE      DROP_USER      TIMESTAMP
ALTER_TABLE          CREATE_TABLE        FUNCTIONS     TRUNCATE
ALTER_TYPE           CREATE_TRIGGER     GRANT        TYPES
ALTER_USER           CREATE_TYPE        INSERT        UPDATE
APPLY                CREATE_USER        INSERT_JSON   USE
ASCII                DATE             INT          UUID
BATCH                DELETE           JSON
BEGIN               DROP_AGGREGATE    KEYWORDS
BLOB                DROP_COLUMNFAMILY LIST_PERMISSIONS
BOOLEAN              DROP_FUNCTION    LIST_ROLES
COUNTER              DROP_INDEX       LIST_USERS
CREATE_AGGREGATE    DROP_KEYSPACE    PERMISSIONS
CREATE_COLUMNFAMILY  DROP_MATERIALIZED_VIEW REVOKE
CREATE_FUNCTION     DROP_ROLE        SELECT
CREATE_INDEX         DROP_TABLE       SELECT_JSON
```

### 3.4.1 Keyspaces

A keyspace is a container to hold application data. It is comparable to a relational database. It is used to group column families together. Typically, a cluster has one keyspace per application. Replication is controlled on a per keyspace basis. Therefore, data that has different replication requirements should reside on different keyspaces.

*Objective:*

Create a keyspace by the name “Students”.

## **Syntax:**

```
CREATE KEYSPACE [IF NOT EXISTS] keyspace_name
  WITH REPLICATION = {
    'class' : 'SimpleStrategy', 'replication_factor' : N }
    | 'class' : 'NetworkTopologyStrategy',
      'dc1_name' : N [, ...]
  }
  [AND DURABLE_WRITES = true|false] ;
```

Keyspace\_name is a user-defined name for the keyspace. It can be a maximum of 48 characters (comprising letters, numbers, and underscores). The first character has to be a letter or a number. If you would like the keyspace\_name to be in uppercase, enclose it within quotes. If a keyspace with the specified name already exists, an error is reported and the operation fails. In order to suppress the error message, use “IF NOT EXISTS”.

When one creates a keyspace, it is required to specify a strategy class. There are two choices available with us. Either we can specify a “SimpleStrategy” or a “NetworkTopologyStrategy” class. While using Cassandra for evaluation purpose (single data center – test and development environment), go with “SimpleStrategy” class and for production usage, work with the “NetworkTopologyStrategy” class. “SimpleStrategy” specifies a simple replication factor for the cluster whereas “NetworkTopologyStrategy” allows to set the replication factor for each cluster independently.

```
DURABLE_WRITES = true|false
```

If DURABLE\_WRITES is set to False, it will bypass the commit log while writing to the keyspace. The default value for Durable\_Writes is TRUE. If SimpleStrategy replication is chosen, do not disable DURABLE\_WRITES.

*Act:*

```
CREATE KEYSPACE Students WITH REPLICATION = {
  'class' : 'SimpleStrategy',
  'replication_factor' : 1
};
```

## **Outcome:**

```
cmd. Command Prompt - cqlsh
cqlsh> CREATE KEYSPACE Students WITH REPLICATION = {
    ...             'class':'SimpleStrategy',
    ...             'replication_factor':1
    ... };
cqlsh>
cqlsh>
```

The replication factor stated above in the syntax for creating keyspace is related to the number of copies of keyspace data that is housed in a cluster.

### *Objective:*

Describe all the existing keyspaces in the cluster.

### Syntax:

```
DESCRIBE KEYSPACES | KEYSPACE <keyspace name>
```

DESCRIBE KEYSPACES generates a list of all keyspaces in the cluster. One can use only “DESC” in lieu of “DESCRIBE”. You can also choose to describe all objects in a specific keyspace (such as “Students”) by using the syntax, “DESC KEYSPACE Students”.

### *Act:*

## DESCRIBE KEYSACES:

### *Outcome:*

```
on Select Command Prompt - cqlsh
cqlsh> DESCRIBE KEYSPACES;
students  system_schema  system_auth  system  system_distributed  system_traces
cqlsh>
```

### *Objective:*

Get more details on the existing keyspaces such as keyspace name, durable writes, strategy class, strategy options, etc.

*Act:*

```
SELECT *
  FROM system.schema_keyspaces;
```

*Outcome:*

```
cqlsh> SELECT * FROM system.schema_keyspaces;
  keyspace_name | durable_writes | strategy_class
+-----+-----+-----+
  demo_con    |      True   | org.apache.cassandra.locator.SimpleStrategy
  system       |      True   | org.apache.cassandra.locator.LocalStrategy
  system_traces |      True   | org.apache.cassandra.locator.SimpleStrategy
  students     |      True   | org.apache.cassandra.locator.SimpleStrategy
+-----+-----+-----+
(4 rows)
```

**Note:** Cassandra converted the Students keyspace to lowercase as quotation marks were not used.

---

*Objective*

Display information about the cluster.

*Act*

```
DESCRIBE CLUSTER;
```

*Outcome:*

---

```
Command Prompt - cqlsh
cqlsh> DESCRIBE CLUSTER;
Cluster: Test Cluster
Partitioner: Murmur3Partitioner
cqlsh>
```

---

*Objective:*

Write the command to use the keyspace “Students”.

*Act:*

```
Use keyspace_name.  
Use connects the client session to the specified keyspace.
```

### *Syntax:*

```
USE Students;
```

### *Outcome:*

---

Command Prompt - cqlsh  
cqlsh> USE Students;  
cqlsh:students>

---

### *Objective:*

Create a column family or table by the name “student\_info”.

### *Act:*

```
CREATE TABLE Student_Info (  
    RollNo int PRIMARY KEY,  
    StudName text,  
    DateofJoining timestamp,  
    LastExamPercent double  
) ;
```

Here “int” data type is a 32-bit signed integer, “text” is a UTF-8 encoded string (Unicode Transformation format that uses 8-bit blocks to represent a character), timestamp is a date and time data type with millisecond precision and double is 64-bit floating point.

### *Outcome:*

```
Command Prompt - cqlsh
cqlsh> USE Students;
cqlsh:students> CREATE TABLE Student_Info (
    ...     RollNo int PRIMARY KEY,
    ...     StudName text,
    ...     DateofJoining timestamp,
    ...     LastExamPercent double
    ... );
cqlsh:students>
cqlsh:students>
```

The table “student\_info” gets created in the keyspace “students”.

**Note:** Tables can have either a single or compound primary key. Always ensure that there is exactly one primary key definition. The primary key, however, can be simple (consisting of a single attribute) or composite (comprising two or more attributes). One cannot change a primary key once it is defined.

RollNo is the basic primary key. The primary key here is a single parameter that identifies the record of a student. The first element in the primary key is a partition key. There are two purpose that a primary key serves: (a) It uniquely identifies a record from a set of records and (b) it determines data locality. When the data is inserted into the cluster, a hash function is applied to the partition key. The output is used to determine the node (or replicas) where the data will be placed. A partition key always belongs to one node.

```
CREATE TABLE user_videos (
    userid uuid,
    added_date timestamp,
    videoid uuid,
    name text,
    preview_image_location text,
    PRIMARY KEY (userid, added_date, videoid)
);
```

```
Command Prompt - cqlsh
cqlsh:students> CREATE TABLE user_videos (
    ...     userid uuid,
    ...     added_date timestamp,
    ...     videoid uuid,
    ...     name text,
    ...     preview_image_location text,
    ...     PRIMARY KEY (userid, added_date, videoid)
    ... );
cqlsh:students>
```

Explanation about the composite PRIMARY KEY:

Primary key (column\_name1, *column\_name2*, *column\_name3* ...)

Primary key ((column\_name4, *column\_name5*), *column\_name6*, *column\_name7* ...)

In the above syntax,

*column\_name1* is the partition key.

*column\_name2* and *column\_name3* are the clustering columns.

*column\_name4* and *column\_name5* are the partitioning keys.

*column\_name6* and *column\_name7* are the clustering columns.

The partition key is used to distribute the data in the table across various nodes that constitute the cluster. The clustering columns are used to specify the order in which the data is arranged inside the partition.

Since the clustering columns specify the order in a single partition, it would be helpful to control the directionality of the sorting by using the “ORDER BY” clause. If one wishes to control the sort order as default of the data model, use “CLUSTERING ORDER BY” clause at the time of creating the table.

```
CREATE TABLE user_videos_1 (
    userid uuid,
    added_date timestamp,
    videoid uuid,
    name text,
    preview_image_location text,
    PRIMARY KEY (userid, added_date, videoid)
) WITH CLUSTERING ORDER BY (added_date DESC, videoid ASC);
```

```
Command Prompt - cqlsh
cqlsh:students> CREATE TABLE user_videos_1 (
    ...     userid uuid,
    ...     added_date timestamp,
    ...     videoid uuid,
    ...     name text,
    ...     preview_image_location text,
    ...     PRIMARY KEY (userid, added_date, videoid)
    ... ) WITH CLUSTERING ORDER BY (added_date DESC, videoid ASC);
cqlsh:students>
```

---

### ***Objective:***

Lookup and display the names of all tables in the current keyspace, or in all the keyspaces if there is no current keyspace.

### ***Act:***

```
DESCRIBE TABLES;
```

### ***Outcome:***

```
Command Prompt - cqlsh
cqlsh:students> DESCRIBE TABLES;
student_info  user_videos_1  user_videos
cqlsh:students>
```

---

### ***Objective:***

Describe the table “student\_info”.

### ***Act:***

```
DESCRIBE TABLE student_info;
```

**Note:** The output is a list of CQL commands with the help of which the table “student\_info” can be recreated. The output is a detailed CQL executable information for a named table (“student\_info” in the example), including materialized views based on the table. It is advisable to check all the settings

before executing the full output as few options are cluster specific, especially in the “WITH” statement.

### ***Outcome:***

```
Command Prompt - cqlsh
cqlsh:students> DESCRIBE TABLE student_info;

CREATE TABLE students.student_info (
    rollno int PRIMARY KEY,
    dateofjoining timestamp,
    lastexampercent double,
    studname text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';

cqlsh:students>
```

---

### ***Objective:***

Insert data into the column family “student\_info”.

The “INSERT” statement is used to insert an entire row or upsert data into an existing row using the full PRIMARY KEY.

An insert writes one or more columns to a record in Cassandra table atomically. An insert statement does not return an output unless IF NOT EXISTS is used. One is not required to place values in all the columns; however, it is mandatory to specify values for all the columns that make up the primary key. The columns that are missing do not occupy any space on disk.

Internally insert and update operations are equal. However, insert does not support counters but update does. Counters will be discussed later in Section 3.5.

### ***Act:***

```

BEGIN BATCH
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExam
Percent)
VALUES (1,'Michael Storm','2012-03-29', 69.6)
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExam
Percent)
VALUES (2,'Stephen Fox','2013-02-27', 72.5)
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExam
Percent)
VALUES (3,'David Flemming','2014-04-12', 81.7)
INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExam
Percent)
VALUES (4,'Ian String','2012-05-11', 73.4)
APPLY BATCH;

```

### ***Outcome:***

```

cq1sh:students> BEGIN BATCH
...     INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
...     VALUES (1,'Michael Storm','2012-03-29',69.6)
...     INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
...     VALUES (2,'Stephen Fox','2013-02-27',72.5)
...     INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
...     VALUES (3,'David Flemming','2014-04-12',81.7)
...     INSERT INTO student_info (RollNo,StudName,DateofJoining,LastExamPercent)
...     VALUES (4,'Ian String','2012-05-11',73.4)
...     APPLY BATCH;
cq1sh:students>

```

A batch (begin batch.... end batch) is used to combine multiple DML (data manipulation statements such as INSERT, UPDATE and DELETE) with the objective of achieving atomicity (either all the DML statements within a batch are executed or none at all) and isolation (client cannot read a partial update from any row unless the complete batch has been successfully updated) for a single partition and ONLY atomicity for multiple partitions.

In the example above, all the four insert statements will succeed or none of it will. Assume there was an error in the third insert statement. In such a case, the entire batch fails. The first two inserts will not apply to the table.

---

### ***Objective:***

Display the data from the table “student\_info”.

### ***Act:***

```
SELECT *
  FROM student_info;
```

The above select statement retrieves data from the “student\_info” table.

***Outcome:***

```
cqlsh:students> select * from student_info;
  rollno | dateofjoining
-----+-----
    1 | 2012-03-29 00:00:00India Standard Time
    2 | 2013-02-27 00:00:00India Standard Time
    4 | 2012-05-11 00:00:00India Standard Time
    3 | 2014-04-12 00:00:00India Standard Time
| lastexampercent | studname
-----+-----
      69.6 | Michael Storm
      72.5 | Stephen Fox
      73.4 | Ian String
      81.7 | David Flemming
(4 rows)
cqlsh:students>
```

---

***Objective:***

Display only those records where the RollNo column has a value either 1 or 2 or 3.

***Act:***

```
SELECT *
  FROM student_info
  WHERE RollNo IN(1,2,3);
```

***Note:*** For the above statement to execute successfully, ensure that the following criteria are satisfied:

1. Either the partition key definition includes the column that is used in the where clause (i.e., search criteria).
2. OR the column being used in the where clause (i.e., search criteria) has an index defined on it using the CREATE INDEX statement.

***Outcome:***

```
cqlsh:students> Select * from student_info where RollNo IN(1,2,3);
  rollno | dateofjoining | lastexampercent | studname
-----+-----+-----+-----+
    1 | 2012-03-29 00:00:00India Standard Time | 69.6 | Michael Storm
    2 | 2013-02-27 00:00:00India Standard Time | 72.5 | Stephen Fox
    3 | 2014-04-12 00:00:00India Standard Time | 81.7 | David Flemming
(3 rows)
cqlsh:students>
```

Let us try running a query with “studname” in the where clause. Since “studname” is neither the primary key column nor a column in the primary key definition and also does not have an index defined on it, such a query will lead to an error.

We set the stage to resolve the error by creating an index on the “studname” column of the “student\_info” table and then subsequently executing the query.

To create an index on the “studname” column of the “student\_info” column family, use

```
CREATE INDEX ON student_info(studname);
```

If data already exists for the column “studname”, then Cassandra indexes the data during the execution of this statement. After the index has been created, Cassandra will index the new data coming into the column automatically.

Cassandra supports creating index on most of the columns barring a counter column. Index can also be created on a primary key column (either a partition key column or a clustering column).

Indexes on the appropriate column or columns can boost up the performance greatly. However, an index on an inappropriate column or columns can have adverse performance impact.

When not to create an index:

1. Do not create an index on a column that is frequently updated or deleted. The reason behind this is that Cassandra stores tombstones in the index until the tombstones limit reaches 100k cells. Once the limit is exceeded, the query involving the indexed column will fail.
2. Do not create an index on low-cardinality column such as the one which stores a Boolean value of “true” and “false”. If an index is created on a low-cardinality column, each value in the index becomes a single row in the index resulting in a huge row for all the false values. Likewise a huge row will be created for all the true values.

- 
3. Do not create an index on a high-cardinality column for a query for a huge volume of records for a small number of results.
- 

**Objective:**

Execute the query using the index defined on “studname” column.

**Act:**

```
SELECT *
  FROM student_info
    WHERE studname='Stephen Fox' ;
```

**Outcome:**

```
cqlsh:students> create index on student_info(studname);
cqlsh:students> select * from student_info where studname='Stephen Fox' ;
rollno | dateofjoining | lastexampercent | studname
-----+-----+-----+-----+
  2 | 2013-02-27 00:00:00India Standard Time |      72.5 | Stephen Fox
(1 rows)
cqlsh:students>
```

---

**Objective:**

Create another index on the “LastExamPercent” column of the “student\_info” column family.

**Act:**

```
CREATE INDEX ON student_info(LastExamPercent);
```

**Outcome:**

```
cqlsh:students> create index on student_info(LastExamPercent);
cqlsh:students> select * from student_info where LastExamPercent = 81.7;
rollno | dateofjoining | lastexampercent | studname
-----+-----+-----+-----+
  3 | 2014-04-12 00:00:00India Standard Time |      81.7 | David Flemming
(1 rows)
```

---

**Objective:**

Determine the number of rows returned in the output using limit.

**Act:**

```
SELECT rollno, studname, dateofjoining, lastexampercent
      FROM student_info LIMIT 2;
```

**Outcome:**

```
Command Prompt - cqlsh
cqlsh:students> SELECT rollno, studname, dateofjoining, lastexampercent FROM student_info LIMIT 2;

 rollno | studname      | dateofjoining           | lastexampercent
-----+-----+-----+-----+
  1 | Michael Storm | 2012-03-28 18:30:00.000000+0000 |      69.6
  2 | Stephen Fox   | 2013-02-26 18:30:00.000000+0000 |      72.5

(2 rows)
cqlsh:students>
```

---

**Objective:**

Determine the number of rows returned in the output using count and limit.

**Act:**

(a) Create a table, “test”.

```
CREATE TABLE test (
  "type" varchar,
  "value" varchar,
  PRIMARY KEY (type,value)
);
```

(b) Insert 5 rows into it.

```
INSERT INTO test(type,value) VALUES('test','tag1');
INSERT INTO test(type,value) VALUES('test','tag2');
INSERT INTO test(type,value) VALUES('test','tag3');
```

```
INSERT INTO test(type,value) VALUES('test','tag4');
INSERT INTO test(type,value) VALUES('test','tag5');
```

(c) Run SELECT \* from test LIMIT 3. It works as expected.

```
Command Prompt - cqlsh
cqlsh:students> SELECT * from test LIMIT 3;

  type | value
-----+
  test | tag1
  test | tag2
  test | tag3

(3 rows)
cqlsh:students>
```

(d) Run the SELECT COUNT(\*) from test LIMIT 3.

### ***Outcome:***

It produces

```
Command Prompt - cqlsh
cqlsh:students> SELECT COUNT(*) from test LIMIT 3;

  count
-----
      5

(1 rows)
```

The output is 3 because of the following reason. First count(\*) is applied to the collection. Since there are five records in the “test” collection, it returns 5. There is only one row returned in the output therefore limit 3 does not have an impact on the output.

---

### ***Objective:***

Use column alias for the column “language” in the “student\_info” table. The column heading should be “knows language”:

**Act:**

```
SELECT rollno, language AS "knows language"  
      FROM student_info;
```

**Outcome:**

```
cqlsh:students> select rollno, language as "knows language" from student_info;  
rollno | knows language  
-----+  
  1 |      ['Hindi, English']  
  4 |      ['Hindi, English']  
  3 | ['Hindi,English,French']  
(3 rows)
```

---

**Objective:**

Update the value held in the “StudName” column of the “student\_info” column family to “David Sheen” for the record where the RollNo column has value = 2.

**Note:** An update updates one or more column values for a given row to the Cassandra table. It does not return anything.

Similar to insert, an UPDATE statement is an upsert operation. If the specified row does not exist, the UPDATE command creates it.

The UPDATE command supports counters. Counters are not supported by INSERT.

**Act:**

```
UPDATE student_info SET StudName = 'David Sheen' WHERE RollNo = 2;
```

**Outcome:**

```
cqlsh:students> UPDATE student_info SET studName = 'David Sheen' WHERE RollNo = 2;
cqlsh:students> select * from student_info where rollno = 2;
+-----+-----+-----+
| rollno | dateofjoining | lastexampercent | studname |
+-----+-----+-----+
| 2 | 2013-02-27 00:00:00India Standard Time | 72.5 | David Sheen |
+-----+
(1 rows)

cqlsh:students>
```

---

### **Objective:**

Update the value of a primary key column.

### **Act:**

```
UPDATE student_info SET rollno=6 WHERE rollno=3;
```

### **Outcome:**

```
Command Prompt - cqlsh
cqlsh:students> UPDATE student_info SET rollno=6 WHERE rollno=3;
InvalidRequest: Error from server: code=2200 [Invalid query] message="PRIMARY KEY part rollno found in SET part"
cqlsh:students>
```

**Note:** It does not allow update to a primary key column.

---

### **Objective:**

To update more than one column of a row of Cassandra table.

### **Act:**

**Step 1:** Before the update

```
SELECT rollno, studname, lastexampercent from student_info where rollno=3;
```

```
cqlsh:students> select rollno, studname, lastexampercent from student_info where rollno=3;
+-----+-----+-----+
| rollno | studname | lastexampercent |
+-----+-----+-----+
| 3 | David Flemming | 81.7 |
+-----+
(1 rows)
```

**Step 2:** Applying the update

```
Update student_info set studname='Samaira', lastexampercent=85 where rollno=3;
```

```
|cqlsh:students> update student_info set studname='Samaira', lastexampercent=85 where rollno=3;
```

**Step 3:** After the update

```
cqlsh:students> select rollno, studname, lastexampercent from student_info where rollno=3;
```

rollno	studname	lastexampercent
3	Samaira	85

(1 rows)

**Objective:**

Delete the column “LastExamPercent” from the “student\_info” table for the record where the RollNo = 2.

**Note:** The Delete statement removes one or more columns from one or more rows of a Cassandra table or removes entire rows if no columns are specified.

**Act:**

```
DELETE LastExamPercent FROM student_info WHERE RollNo=2;
```

**Outcome:**

```
cqlsh:students> DELETE LastExamPercent FROM student_info where RollNo=2;
cqlsh:students> select * from student_info where rollno = 2;


| rollno | dateofjoining                          | lastexampercent | studname    |
|--------|----------------------------------------|-----------------|-------------|
| 2      | 2013-02-27 00:00:00India Standard Time | null            | David Sheen |



(1 rows)


cqlsh:students>
```

**Objective:**

Delete a row (where the RollNo = 2) from the table “student\_info”.

**Act:**

```
DELETE FROM student_info WHERE RollNo=2;
```

**Outcome:**

```
cqlsh:students> DELETE FROM student_info where RollNo=2;
cqlsh:students> select * from student_info where rollno=2;
(0 rows)
cqlsh:students>
```

---

**Objective:**

Create a table “project\_details” with primary key as (project\_id, project\_name).

**Act:**

```
CREATE TABLE project_details (
    project_id int,
    project_name text,
    stud_name text,
    rating double,
    duration int,
    PRIMARY KEY (project_id, project_name));
```

**Outcome:**

```
cqlsh:students> CREATE TABLE project_details (
    ... project_id int,
    ... project_name text,
    ... stud_name text,
    ... rating double,
    ... duration int,
    ... PRIMARY KEY (project_id, project_name));
cqlsh:students>
```

---

**Objective:**

Insert data into the column family “project\_details”.

**Act:**

```

BEGIN BATCH
INSERT INTO project_details (project_id,project_name,stud_name, rating, duration) VALUES (1,'MS data migration', 'David Sheen', 3.5,720)
INSERT INTO project_details (project_id,project_name,stud_name, rating, duration) VALUES (1,'MS Data Warehouse', 'David Sheen', 3.9,1440)
INSERT INTO project_details (project_id,project_name,stud_name, rating, duration) VALUES (2,'SAP Reporting', 'Stephen Fox', 4.2,3000)
INSERT INTO project_details (project_id,project_name,stud_name, rating, duration) VALUES (2,'SAP BI DW', 'Stephen Fox', 4,9000)
APPLY BATCH;

```

### ***Outcome:***

```

cqlsh:students> BEGIN BATCH
...   INSERT INTO project_details (project_id,project_name,stud_name, rating, duration)
...   VALUES (1,'MS data Migration', 'David Sheen', 3.5,720)
...   INSERT INTO project_details (project_id,project_name,stud_name, rating, duration)
...   VALUES (1,'MS Data Warehouse', 'David Sheen', 3.9,1440)
...   INSERT INTO project_details (project_id,project_name,stud_name, rating, duration)
...   VALUES (2,'SAP Reporting', 'Stephen Fox', 4.2,3000)
...   INSERT INTO project_details (project_id,project_name,stud_name, rating, duration)
...   VALUES (2,'SAP BI DW', 'Stephen Fox', 4,9000)
... APPLY BATCH;

```

---

### ***Objective:***

Display all the rows of the “project\_details” table.

### ***Act:***

```

SELECT *
  FROM project_details;

```

### ***Outcome:***

```

cqlsh:students> select * from project_details;

```

project_id	project_name	duration	rating	stud_name
1	MS Data Warehouse	1440	3.9	David Sheen
1	MS data Migration	720	3.5	David Sheen
2	SAP BI DW	9000	4	Stephen Fox
2	SAP Reporting	3000	4.2	Stephen Fox

(4 rows)

Now we update a row in a table with a compound primary key.

In order to specify a row, the WHERE clause must provide a value for each column of the row’s primary key. To specify more than one row, you can use

the IN keyword to introduce a list of possible values. You can only do this for the last column of the primary key.

To specify more than one row, use primary\_key\_name IN (primary\_key\_value,primary\_key\_value ... ). This only works for the last component of the primary key.

---

***Objective:***

Display row/record from the “project\_details” table wherein the project\_id=1;

***Act:***

```
SELECT *
  FROM project_details
 WHERE project_id=1;
```

***Outcome:***

```
cqlsh:students> Select * from project_details where project_id=1;
project_id | project_name      | duration | rating | stud_name
-----+-----+-----+-----+-----+
      1 | MS Data Warehouse | 1440    | 3.9   | David Sheen
      1 | MS data Migration | 720     | 3.5   | David Sheen
(2 rows)
```

---

***Objective:***

Use “allow filtering” with the Select statement.

**Note:** When one attempts a potentially expensive query that might involve searching a range of rows, a prompt such as the one shown below appears:

Bad Request: Cannot execute this query as it might involve data filtering and thus may have unpredictable performance. If you want to execute this query despite the performance unpredictability, use ALLOW FILTERING.

***Act:***

```
SELECT *
  FROM project_details
    WHERE project_name='MS Data Warehouse' ALLOW FILTERING;
```

### ***Outcome:***

```
cqlsh:students> Select * from project_details where project_name='MS Data Warehouse' allow filtering;
project_id | project_name      | duration | rating | stud_name
-----+-----+-----+-----+-----+
  1 | MS Data Warehouse | 1440    | 3.9   | David Sheen
(1 rows)
```

---

### ***Objective:***

Sort or order the rows/records of the “project\_details” column in ascending order of project\_name.

### ***Act:***

```
SELECT *
  FROM project_details
    WHERE project_id IN (1,2)
        ORDER BY project_name DESC;
```

### ***Outcome:***

```
cqlsh:students> SELECT * FROM project_details WHERE project_id IN (1,2) ORDER BY project_name DESC;
project_id | project_name      | duration | rating | stud_name
-----+-----+-----+-----+-----+
  2 | SAP Reporting     | 3000    | 4.2   | Stephen Fox
  2 | SAP BI DW         | 9000    | 4      | Stephen Fox
  1 | MS data Migration | 720     | 3.5   | David Sheen
  1 | MS Data Warehouse | 1440    | 3.9   | David Sheen
(4 rows)
```

***Note:*** By default, sorting or ordering is done in ascending order. The user can specify the order by using the keyword “ASC” for ascending or “DESC” for descending.

ORDER BY clause can select a single column only. This column is the second column of the compound Primary key. This applies even when the compound Primary key has more than two columns.

*When specifying the ORDER BY clause, use only the column name and not the column alias.*

## 3.5 COLLECTIONS

---

### Picture This

You are required to store details about users of service “xyz”. The details of the user include User\_ID, User\_Name, User\_Contact\_Nos, User\_Email\_Ids. A user may have  $n$  number of Contact Nos and also may have  $n$  number of Email IDs. How do we accomplish this task in RDBMS?

We would create a table, let us say “Users”, to store details such as “User\_ID”, “User\_Name” and another table “UsersContactDetails”. The relationship between “UsersContactDetails” and “Users” is many to one. Likewise, we would create a table “UsersEmailIDs” and establish a many to one relationship between “UsersEmailIDs” and the “Users” table.

However, the multiple Contact Nos and multiple Email IDs problem can be solved by defining a column as a collection. The usage of collection types for columns is not only convenient but intuitive as well.

### ***When to use collection?***

Use collection when it is required to store or denormalize a small amount of data.

### ***What is the limit on the values of items in a collection?***

The values of items in a collection are limited to 64K.

### ***Where to use collections?***

Collections can be used when you need to store the following:

1. Phone numbers of users
2. Email ids of users

### ***When should one refrain from using a collection?***

One should refrain from using a collection when the data has unbound growth potential such as all the messages posted by a user or all the event data as captured by a sensor. When faced with such a situation, use a table with compound primary key with data being held in clustering columns.

### ***Collection Types***

CQL makes use of the following collection types:

- 1. Set collection:** A column of type set consists of unordered unique values. However, when the column is queried, it returns the values in sorted order. For example, for text values, it sorts in alphabetical order.
- 2. List collection:** When the order of elements matter, one should opt for a list collection. For example, when you store the preferences of places to visit by a user, you would like to respect his/her preferences and retrieve the values in the order in which he/she has entered rather than in sorted order. A list also allows one to store the same value multiple times.
- 3. Map collection:** As the name implies, a map is used to map one thing to another. A map is a pair of typed values. It is used to store time-stamp related information. Each element of the map is stored as a Cassandra column. Each element can be individually queried, modified and deleted.

***Objective:***

Alter the schema for the table “student\_info” to add a column hobbies.

***Act:***

```
ALTER TABLE student_info ADD hobbies set<text>;
```

***Output:***

```
cqlsh:students> ALTER TABLE student_info ADD hobbies set<text>;
```

Confirm the structure of the table after the change has been made.

```
cqlsh:students> describe table student_info;
CREATE TABLE student_info (
    rollno int,
    dateofjoining timestamp,
    hobbies set<text>,
    lastexampercent double,
    studname text,
    PRIMARY KEY (rollno)
) WITH
    bloom_filter_fp_chance=0.010000 AND
    caching='KEYS_ONLY' AND
    comment='' AND
    dclocal_read_repair_chance=0.000000 AND
    gc_grace_seconds=864000 AND
    index_interval=128 AND
    read_repair_chance=0.100000 AND
    replicate_on_write='true' AND
    populate_io_cache_on_flush='false' AND
    default_time_to_live=0 AND
    speculative_retry='NONE' AND
    memtable_flush_period_in_ms=0 AND
    compaction={'class': 'SizeTieredCompactionStrategy'} AND
    compression={'sstable_compression': 'LZ4Compressor'};

CREATE INDEX student_info_lastexampercent_idx ON student_info (lastexampercent);

CREATE INDEX student_info_studname_idx ON student_info (studname);
```

---

### ***Objective:***

Alter the schema of the table “student\_info” to add a list column “language”.

### ***Act:***

```
ALTER TABLE student_info ADD language list<text>;
```

### ***Outcome:***

```
cqlsh:students> ALTER TABLE student_info ADD language list<text>;
cqlsh:students>
```

Confirm the structure of the table after the change has been made.

```
cqlsh:students> describe table student_info;
CREATE TABLE student_info (
  rollno int,
  dateofjoining timestamp,
  hobbies set<text>,
  language list<text>,
  lastexampercent double,
  studname text,
  PRIMARY KEY (rollno)
) WITH
bloom_filter_fp_chance=0.010000 AND
caching='KEYS_ONLY' AND
comment='' AND
dclocal_read_repair_chance=0.000000 AND
gc_grace_seconds=864000 AND
index_interval=128 AND
read_repair_chance=0.100000 AND
replicate_on_write='true' AND
populate_io_cache_on_flush='false' AND
default_time_to_live=0 AND
speculative_retry='NONE' AND
memtable_flush_period_in_ms=0 AND
compaction={'class': 'SizeTieredCompactionStrategy'} AND
compression={'sstable_compression': 'LZ4Compressor'};

CREATE INDEX student_info_lastexampercent_idx ON student_info (lastexampercent);
CREATE INDEX student_info_studname_idx ON student_info (studname);
cqlsh:students>
```

---

### ***Objective:***

Update the table “student\_info” to provide the values for “hobbies” for the student with Rollno=1.

### ***Act:***

```
UPDATE student_info
  SET hobbies = hobbies + {'Chess, Table Tennis'}
  WHERE RollNo=1;
```

### ***Outcome:***

```
cqlsh:students> UPDATE student_info
...   SET hobbies = hobbies + {'Chess, Table Tennis'} WHERE RollNo=1;
```

To confirm the values in the hobbies column, use the below command:

```
SELECT *
  FROM student_info
 WHERE RollNo=1;
```

```
cqlsh:students> select * from student_info where RollNo=1;
 rollno | dateofjoining | hobbies | language | lastexampercent | studname
-----+-----+-----+-----+-----+-----+
 1 | 2012-03-29 00:00:00India Standard Time | {'Chess, Table Tennis'} | null | 69.6 | Michael Storm
(1 rows)
```

Likewise update a few more records:

```
cqlsh:students> UPDATE student_info
...      SET hobbies = hobbies + {'Chess, Badminton'} WHERE RollNo=3;
cqlsh:students> UPDATE student_info
...      SET hobbies = hobbies + {'Lawn Tennis, Table Tennis, Golf'} WHERE RollNo=4;
cqlsh:students>
```

Records after the updation:

```
cqlsh:students> select * from student_info;
 rollno | dateofjoining | hobbies | language | lastexampercent | studname
-----+-----+-----+-----+-----+-----+
 1 | 2012-03-29 00:00:00India Standard Time | {'Chess, Table Tennis'} | null | 69.6 | Michael Storm
 4 | 2012-05-11 00:00:00India Standard Time | {'Lawn Tennis, Table Tennis, Golf'} | null | 73.4 | ...
 3 | 2014-04-12 00:00:00India Standard Time | {'Chess, Badminton'} | null | 81.7 | David
(3 rows)
```

### ***Objective:***

Update values in the list column “language” of the table “student\_info”.

### ***Act:***

```
UPDATE student_info
      SET language = language + ['Hindi, English']
 WHERE RollNo=1;
```

### ***Outcome:***

```
cqlsh:students> UPDATE student_info
...      SET language = language + ['Hindi, English'] WHERE RollNo=1;
cqlsh:students>
```

Likewise update the remaining records.

```
cqlsh:students> UPDATE student_info
...      SET language = language + ['Hindi,English,French'] WHERE RollNo=3;
cqlsh:students> UPDATE student_info
...      SET language = language + ['Hindi, English'] WHERE RollNo=4;
cqlsh:students>
```

To view the updates to the records, use the below statement:

```
cqlsh:students> select rollNo, studname, hobbies, language from student_info;
 rollno | studname      | hobbies                  | language
-----+-----+-----+-----+
  1 | Michael Storm | {'Chess, Table Tennis'} | ['Hindi, English']
  4 | Ian String    | {'Lawn Tennis, Table Tennis, Golf'} | ['Hindi, English']
  3 | David Flemming | {'Chess, Badminton'} | ['Hindi,English,French']
(3 rows)
```

### 3.5.1 More Practice on Collections (SET and LIST)

***Objective:***

Create a table “users” with an “emails” column. The type of this column “emails” should be “set”.

***Act:***

```
CREATE TABLE users (
    user_id text PRIMARY KEY,
    first_name text,
    last_name text,
    emails set<text>
);
```

***Outcome:***

```
cqlsh:students> CREATE TABLE users (
...      user_id text PRIMARY KEY,
...      first_name text,
...      last_name text,
...      emails set<text>
... );
cqlsh:students>
```

***Objective:***

Insert values into the “emails” column of the “users” table.

**Note:** Set values must be unique.

**Act:**

```
INSERT INTO users
(user_id, first_name, last_name, emails)
    VALUES('AB', 'Albert', 'Baggins', {'a@baggins.com', 'baggins@gmail.com'});
```

**Outcome:**

```
cqlsh:students> INSERT INTO users (user_id, first_name, last_name, emails)
...     VALUES('AB', 'Albert', 'Baggins', {'a@baggins.com', 'baggins@gmail.com'});
```

---

**Objective:**

Add an element to a set using the UPDATE command and the addition (+) operator.

**Act:**

```
UPDATE users
    SET emails = emails + {'ab@friendsofmordor.org'}
    WHERE user_id = 'AB';
```

**Outcome:**

```
cqlsh:students> UPDATE users
...     SET emails = emails + {'ab@friendsofmordor.org'} WHERE user_id = 'AB';
cqlsh:students>
```

---

**Objective:**

Retrieve the email addresses for Albert from the “users” set.

**Act:**

```
SELECT user_id, emails
    FROM users
    WHERE user_id = 'AB';
```

**Outcome:**

```
cqlsh:students> SELECT user_id, emails FROM users WHERE user_id = 'AB';
  user_id | emails
  -----+-----
    AB  | {'a@baggins.com', 'ab@friendsofmordor.org', 'baggins@gmail.com'}
(1 row)
```

---

***Objective:***

Remove an element from a set using the subtraction (-) operator.

***Act:***

```
UPDATE users
  SET emails = emails - {'ab@friendsofmordor.org'}
 WHERE user_id = 'AB';
```

***Outcome:***

```
cqlsh:students> UPDATE users
  ...   SET emails = emails - {'ab@friendsofmordor.org'} WHERE user_id = 'AB';
```

To view the records from the “users” table:

```
cqlsh:students> select * from users;
  user_id | emails           | first_name | last_name
  -----+-----+-----+-----+
    AB  | {'a@baggins.com', 'baggins@gmail.com'} | Albert    | Baggins
(1 rows)
```

---

***Objective:***

Remove all elements from a set by using the UPDATE or DELETE statement.

***Act:***

```
UPDATE users
  SET emails = {}
 WHERE user_id = 'AB';
```

```
cqlsh:students> UPDATE users SET emails = {} WHERE user_id = 'AB';
```

***Outcome:***

The above command removes the emails column. Here is the confirmation:

```
cqlsh:students> select * from users;
+-----+-----+-----+-----+
| user_id | emails | first_name | last_name |
+-----+-----+-----+-----+
| AB     | null   | Albert    | Baggins   |
+-----+-----+-----+-----+
(1 rows)
```

Or

```
DELETE emails
  FROM users
 WHERE user_id = 'AB';
```

```
|cqlsh:students> DELETE emails FROM users WHERE user_id = 'AB';
```

The above command removes the emails column. Here is the confirmation:

```
cqlsh:students> select * from users;
+-----+-----+-----+-----+
| user_id | emails | first_name | last_name |
+-----+-----+-----+-----+
| AB     | null   | Albert    | Baggins   |
+-----+-----+-----+-----+
(1 rows)
```

---

***Objective:***

Alter the “users” table to add a column “top\_places” of type list.

***Act:***

```
ALTER TABLE users ADD top_places list<text>;
```

```
|cqlsh:students> ALTER TABLE users ADD top_places list<text>;
```

---

***Objective:***

Update the list column “top\_places” in the “users” table for user\_id = ‘AB’.

***Act:***

```
UPDATE users
    SET top_places = [ 'Lonavla', 'Khandala' ]
        WHERE user_id = 'AB';
```

```
cqlsh:students> UPDATE users
...     SET top_places = [ 'Lonavla', 'Khandala' ] WHERE user_id = 'AB';
cqlsh:students>
```

---

### ***Objective:***

Prepend an element to the list by enclosing it in square brackets and using the addition (+) operator.

### ***Act:***

```
UPDATE users
    SET top_places = [ 'Mahabaleshwar' ] + top_places
        WHERE user_id = 'AB';

cqlsh:students> UPDATE users
...     SET top_places = [ 'Mahabaleshwar' ] + top_places WHERE user_id = 'AB';
```

### ***Outcome:***

```
cqlsh:students> select * from users;
user_id | emails | first_name | last_name | top_places
-----+-----+-----+-----+-----
AB | null | Albert | Baggins | ['Mahabaleshwar', 'Lonavla', 'Khandala']
(1 rows)
```

---

### ***Objective:***

Append an element to the list by switching the order of the new element data and the list name in the update command.

### ***Act:***

```
UPDATE users
    SET top_places = top_places + [ 'Tapola' ]
        WHERE user_id = 'AB';
```

```
cqlsh:students> UPDATE users
...     SET top_places = top_places + [ 'Tapola' ] WHERE user_id = 'AB';
cqlsh:students>
```

**Outcome:**

```
cqlsh:students> select * from users;
  user_id | emails | first_name | last_name | top_places
-----+-----+-----+-----+-----+
    AB |  null |    Albert |   Baggins | ['Mahabaleshwar', 'Lonavla', 'Khandala', 'Tapola']
(1 rows)
```

---

**Objective:**

Query the database for a list of top places.

**Act:**

```
SELECT user_id, top_places
  FROM users
 WHERE user_id = 'AB';
```

**Outcome:**

```
cqlsh:students> SELECT user_id, top_places FROM users WHERE user_id = 'AB';
  user_id | top_places
-----+-----+
    AB | ['Mahabaleshwar', 'Lonavla', 'Khandala', 'Tapola']
(1 rows)
```

---

**Objective:**

Remove an element from a list using the DELETE command and the list index position in square brackets.

The record as it exists prior to deletion:

```
cqlsh:students> SELECT user_id, top_places FROM users WHERE user_id = 'AB';
  user_id | top_places
-----+-----+
    AB | ['Mahabaleshwar', 'Lonavla', 'Khandala', 'Tapola']
(1 rows)
```

**Act:**

```
DELETE top_places[3]
  FROM users
    WHERE user_id = 'AB';
```

```
|cqlsh:students> DELETE top_places[3] FROM users WHERE user_id = 'AB';
```

### ***Outcome:***

The status after deletion:

```
cqlsh:students> select * from users;
user_id | emails | first_name | last_name | top_places
-----+-----+-----+-----+-----
  AB | null |    Albert |   Baggins | ['Mahabaleshwar', 'Lonavla', 'Khandala']
(1 rows)
```

## ***Using Map:*** Key–Value Pair

---

### ***Objective:***

Alter the “users” table to add a map column “todo”.

### ***Act:***

A map is a collection type that is used to store key–value pairs. As the name implies, it maps one thing to another.

**Example:** The requirement is to store a list of pre-requisite courses for each course.

Assume that the course is “Big Data and Analytics” and the pre-requisite courses are “overview of big data technology landscape” and “digital landscape”. For this, the most ideal collection data type is the map data type.

### ***Syntax:***

```
Create table Courses {id int, Prereq map<text, text>};
```

To insert values into the table:

```
Insert into Courses(id, Prereq) values (1, {'Big Data and Analytics': 'Overview of big data technology landscape', 'AI and Automation': 'Digital Landscape'});
```

Now to achieve the objective of adding a map column “todo” to an already existing “users” table, the command is:

```
ALTER TABLE users
    ADD todo map<timestamp, text>;
```

```
|cqlsh:students> ALTER TABLE users ADD todo map<timestamp, text>;
```

**Outcome:**

```
cqlsh:students> describe table users;

CREATE TABLE users (
    user_id text,
    emails set<text>,
    first_name text,
    last_name text,
    todo map<timestamp, text>,
    top_places list<text>,
    PRIMARY KEY (user_id)
) WITH
    bloom_filter_fp_chance=0.010000 AND
    caching='KEYS_ONLY' AND
    comment='' AND
    dclocal_read_repair_chance=0.000000 AND
    gc_grace_seconds=864000 AND
    index_interval=128 AND
    read_repair_chance=0.100000 AND
    replicate_on_write='true' AND
    populate_io_cache_on_flush='false' AND
    default_time_to_live=0 AND
    speculative_retry='NONE' AND
    memtable_flush_period_in_ms=0 AND
    compaction={'class': 'SizeTieredCompactionStrategy'} AND
    compression={'sstable_compression': 'LZ4Compressor'};

cqlsh:students>
```

---

**Objective:**

Update the record for user (user\_id = ‘AB’) in the “users” table.

**Act:**

```
The record from user_id = 'AB' as it exists in the "users" table
```

```
cqlsh:students> select * from users where user_id='AB';
  user_id | emails | first_name | last_name | todo | top_places
-----+-----+-----+-----+-----+-----+
    AB |    null |    Albert |    Baggins | null | ['Mahabaleshwar', 'Lonavla', 'Khandala']
(1 rows)
```

```
cqlsh:students> UPDATE users
    ...      SET todo =
    ...      ... {'2014-9-24': 'Cassandra Session',
    ...      ...     '2014-10-2 12:00' : 'MongoDB Session'}
    ...      WHERE user_id = 'AB';
```

### ***Outcome:***

```
cqlsh:students> select user_id, todo from users where user_id='AB';
  r_id | todo
-----+-----+
    AB | {'2014-09-24 00:00:00India Standard Time': 'Cassandra Session', '2014-10-02 12:00:00India Standard Time': 'MongoDB Se
```

---

### ***Objective:***

Delete an element from the map using the DELETE command and enclose the timestamp of the element in square brackets.

### ***Act:***

```
DELETE todo['2014-9-24']
  FROM users
    WHERE user_id = 'AB';
```

```
cqlsh:students> DELETE todo['2014-9-24'] FROM users WHERE user_id = 'AB';
```

### ***Outcome:***

```
cqlsh:students> select user_id, todo from users where user_id='AB';
  user_id | todo
-----+-----+
    AB | {'2014-10-02 12:00:00India Standard Time': 'MongoDB Session'}
(1 rows)
```

---

## **3.6 CASSANDRA COUNTER COLUMN**

A counter is a special column that is changed in increments. For example, we may need a counter column to count the number of times a particular book is issued from the library by the students.

### Step 1:

```
CREATE TABLE library_book (
    counter_value counter,
    book_name varchar,
    stud_name varchar,
    PRIMARY KEY (book_name, stud_name)
);
```

```
cqlsh:students> CREATE TABLE library_book
    ...  (
    ...      counter_value counter,
    ...      book_name varchar,
    ...      stud_name varchar,
    ...      PRIMARY KEY (book_name, stud_name)
    ... );
```

### Step 2: Load data into the counter column.

```
UPDATE library_book
    SET counter_value = counter_value + 1
        WHERE book_name='Fundamentals of Business Analytics' AND stud_
name='jeet';
```

```
cqlsh:students> UPDATE library_book
    ...  SET counter_value = counter_value + 1
    ... WHERE book_name='Fundamentals of Business Analytics' AND stud_name='jeet';
```

Take a look at the counter value.

```
SELECT *
    FROM library_book;
```

The output is

```
cqlsh:students> select * from library_book;
book_name          | stud_name | counter_value
-----+-----+-----+
Fundamentals of Business Analytics |      jeet |      1
(1 rows)
```

Let us increase the value of the counter.

```
UPDATE library_book
  SET counter_value = counter_value + 1
    WHERE book_name='Fundamentals of Business Analytics' AND stud_
name='shaan';
```

```
cqlsh:students> UPDATE library_book
  ... SET counter_value = counter_value + 1
  ... WHERE book_name='Fundamentals of Business Analytics' AND stud_name='shaan';
```

Again, take a look at the counter value.

```
cqlsh:students> select * from library_book;
book_name          | stud_name | counter_value
-----+-----+-----+
Fundamentals of Business Analytics |    jeet    |      1
Fundamentals of Business Analytics |    shaan    |      1
(2 rows)
```

Update another record for Stud\_name "Jeet".

```
UPDATE library_book
  SET counter_value = counter_value + 1
    WHERE book_name='Fundamentals of Business Analytics' AND stud_
name='jeet';
```

```
cqlsh:students> UPDATE library_book
  ... SET counter_value = counter_value + 1
  ... WHERE book_name='Fundamentals of Business Analytics' AND stud_name='jeet';
```

Let us take a look at the counter value, one last time.

```
cqlsh:students> select * from library_book;
book_name          | stud_name | counter_value
-----+-----+-----+
Fundamentals of Business Analytics |    jeet    |      2
Fundamentals of Business Analytics |    shaan    |      1
(2 rows)
```

## 3.7 TIME-TO-LIVE (TTL)

Time-to-live (TTL) is used to expire data in a column or table. Columns and tables support an optional expiration period called TTL (time-to-live); TTL is not supported on counter columns. The TTL value is specified in seconds.

### **Demonstration:**

**Step 1:** Create a table “userlogin”.

```
CREATE TABLE userlogin(
    userid int primary key, password text
) ;
```

```
|cq1sh:students> CREATE TABLE userlogin(
|... userid int primary key, password text
|... );
```

**Step 2:** Insert values into the table “userlogin” using TTL.

```
INSERT INTO userlogin (userid, password)
    VALUES (1, 'infy') USING TTL 30;
```

```
|cq1sh:students> INSERT INTO userlogin (userid, password)
|... VALUES (1, 'infy') USING TTL 30;
```

```
SELECT TTL (password)
    FROM userlogin
    WHERE userid=1;
```

```
|cq1sh:students> SELECT TTL (password)
|... FROM userlogin
|... WHERE userid=1;

|ttl(password)
|-----
|18
|(1 rows)
```

## **3.8 ALTER COMMANDS**

---

Alter commands are used to alter the structure/schema of an existing table such as adding a column, dropping a column, changing the data type of an existing column etc.

1. Create a table “sample” with columns “sample\_id” and “sample\_name”.

```
CREATE TABLE sample(
    sample_id text,
    sample_name text,
    PRIMARY KEY (sample_id)
);
```

```
cqlsh:students> Create table sample (sample_id text, sample_name text, primary key (sample_id));
cqlsh:students>
```

2. Insert a record into the table “sample”.

```
INSERT INTO sample(
    sample_id, sample_name)
    VALUES ('S101', 'Big Data');
```

```
cqlsh:students> Insert into sample(sample_id, sample_name) values( 'S101', 'Big Data' );
```

3. View the records of the table “sample”.

```
SELECT *
    FROM sample;
```

```
cqlsh:students> select * from sample;
sample_id | sample_name
-----+-----
S101    |    Big Data
(1 rows)
```

4. Alter the schema of the table “sample”. Change the data type of the column “sample\_id” to integer from text.

```
ALTER TABLE sample
    ALTER sample_id TYPE int;
```

```
cqlsh:students> alter table sample alter sample_id type int;
```

After the data type of the column “sample\_id” is changed from text to integer, try inserting a record as follows and observe the error message:

```
INSERT INTO sample(sample_id, sample_name)
    VALUES( 'S102', 'Big Data' );
```

```
cqlsh:students> Insert into sample(sample_id, sample_name) values( 'S102', 'Big Data' );
Bad Request: Invalid STRING constant (S102) for sample_id of type int
cqlsh:students>
```

Try inserting a record as given below into the table “sample”.

```
INSERT INTO sample(sample_id, sample_name)
    VALUES( 102, 'Big Data' );
```

```
cqlsh:students> Insert into sample(sample_id, sample_name) values( 102, 'Big Data' );
cqlsh:students> select * from sample;
sample_id | sample_name
-----+-----
 1395732529 |    Big Data
      102 |    Big Data
(2 rows)
```

5. Alter the data type of the “sample\_id” column to varchar from integer.

```
ALTER TABLE sample
    ALTER sample_id TYPE varchar;
```

```
cqlsh:students> alter table sample alter sample_id type varchar;
```

Check the records after the data type of “sample\_id” has been changed to varchar from integer.

```
cqlsh:students> select * from sample;
sample_id | sample_name
-----+-----
      S101 |    Big Data
\x00\x00\x00f |    Big Data
(2 rows)
```

6. Drop the column “sample\_id” from the table “sample”.

```
ALTER TABLE sample
    DROP sample_id;
```

```
cqlsh:students> alter table sample drop sample_id;
Bad Request: Cannot drop PRIMARY KEY part sample_id
```

**Note:** The request to drop the “sample\_id” column from the table “sample” does not succeed, as it is the primary key column.

Drop the column “sample\_name” from the table “sample”.

```
ALTER TABLE sample
  DROP sample_name;
```

```
|cq1sh:students> alter table sample drop sample_name;
```

**Note:** The above request to drop the column “sample\_name” from table “sample” succeeds.

7. Drop the column family/table “sample”.

```
DROP columnfamily sample;
```

```
|cdjsu:studenfs> qrob cojounfsmijλ səmbje:
```

The above request succeeds. The table/column family no longer exists in the keyspace.

Confirm the non-existence of the table “sample” in the keyspace by giving the following command:

```
cq1sh:students> describe table sample;
```

```
Column family 'sample' not found
```

8. Drop the keyspace “students”.

```
DROP keyspace students;
```

```
|cq1sh:students> drop keyspace students;
```

Confirm the non-existence of the keyspace “students” by issuing the following command:

```
cq1sh:students> describe keyspace students;
```

```
Keyspace 'students' not found.
```

## 3.9 IMPORT FROM AND EXPORT TO CSV

---

### 3.9.1 Export to CSV

**Objective :** Export the contents of the table/column family “elearninglists” present in the “students” database to a CSV file (d:\elearninglists.csv).

*Act:*

**Step 1:** Check the records of the table “elearninglists” present in the “students” database.

```
SELECT *
  FROM elearninglists;
```

```
cqlsh:students> select * from elearninglists;
  id | course_order | course_id | courseowner | title
-----+-----+-----+-----+-----+
  101 |          1 |    1001 | Subhashini | NoSQL Cassandra
  101 |          2 |    1002 | Seema      | NoSQL MongoDB
  101 |          3 |    1003 | Seema      | Hadoop Sqoop
  101 |          4 |    1004 | Subhashini | Hadoop Flume
(4 rows)
```

**Step 2:** Execute the below command at the cqlsh prompt:

```
COPY elearninglists (id, course_order, course_id, courseowner, title) TO
'd:\elearninglists.csv';
```

```
cqlsh:students> copy elearninglists (id, course_order, course_id, courseowner, title) to 'd:\elearninglists.csv';
4 rows exported in 0.000 seconds.
cqlsh:students>
```

**Step 3:** Check the existence of the “elearninglists.csv” file in the “D:\”. Given below is the content of the “d:\elearninglists.csv” file.

	A	B	C	D	E
1	101		1	1001	Subhashini
2	101		2	1002	Seema
3	101		3	1003	Seema
4	101		4	1004	Subhashini
					Hadoop Flume

### 3.9.2 Import from CSV

#### **Objective:**

To import data from “D:\elearninglists.csv” into the table “elearninglists” present in the “students” database.

**Step 1:** Check for the table “elearninglists” in the “students” database. If the table is already present, truncate the table. This will remove all records from the table but retain the structure of the table.

In our case, the table “elearninglists” is already present in the “students” database. Let us take a look at the records of the “elearninglists” before we run the truncate command on it.

```
cqlsh:students> select * from elearninglists;
+-----+-----+-----+-----+-----+
| id   | course_order | course_id | courseowner | title
+-----+-----+-----+-----+-----+
| 101  |           1 |    1001  | Subhashini  | NoSQL Cassandra
| 101  |           2 |    1002  | Seema       | NoSQL MongoDB
| 101  |           3 |    1003  | Seema       | Hadoop Sqoop
| 101  |           4 |    1004  | Subhashini  | Hadoop Flume
+-----+-----+-----+-----+-----+
(4 rows)
```

Truncate the table using the below command:

```
TRUNCATE elearninglists;
```

```
cqlsh:students> Truncate elearninglists;
cqlsh:students>
```

**Note:** No record is present in the table “elearninglists”. The structure/schema is, however, preserved. We confirm it by executing the below command at the cqlsh prompt:

```
cqlsh:students> select * from elearninglists;
(0 rows)
cqlsh:students>
```

**Step 2:** Check for the content of the “D:\elearninglists.csv” file

	A	B	C	D	E
1	101		1	1001	Subhashini
2	101		2	1002	Seema
3	101		3	1003	Seema
4	101		4	1004	Subhashini
					Hadoop Flume

**Note:** The content in the CSV agrees with the structure of the table “elearninglists” in the “students” database. The structure should be such that the content from the CSV can be housed within it without any issues.

**Step 3:** Execute the below command to import data from “d:\elearninglists.csv” into the table “elearninglists” in the database “students”.

```
COPY elearninglists (id, course_order, course_id, courseowner, title) FROM
'd:\elearninglists.csv';
```

```
cqlsh:students> copy elearninglists (id, course_order, course_id, courseowner, title) from 'd:\elearninglists.csv';
4 rows imported in 0.031 seconds.
cqlsh:students>
```

**Step 4:** Confirm that records have been imported into the table.

```
SELECT *
  FROM elearninglists;
```

```
cqlsh:students> select * from elearninglists;

```

id	course_order	course_id	courseowner	title
101	1	1001	Subhashini	NoSQL Cassandra
101	2	1002	Seema	NoSQL MongoDB
101	3	1003	Seema	Hadoop Sqoop
101	4	1004	Subhashini	Hadoop Flume

```
(4 rows)
```

```
cqlsh:students>
```

### 3.9.3 Import from STDIN

***Objective:***

Import data into an existing table, “persons” present in the “students” database. The data is to be provided by the user using the standard input device.

**Step 1:** Ensure that the table “persons” exists in the database “students”.

```
DESCRIBE TABLE persons;
```

```
cqlsh:students> describe table persons;
```

```
CREATE TABLE persons (
  id int,
  fname text,
  lname text,
  PRIMARY KEY (id)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='NONE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};
```

```
cqlsh:students>
```

**Step 2:** Import data into an existing table “persons” present in the “students” database. Data is to be provided by the user using the standard input device.

```
COPY persons (id, fname, lname) FROM STDIN;
```

```
cqlsh:students> COPY persons (id, fname, lname) FROM STDIN;  
[Use \. on a line by itself to end input]  
[copy] 1,"Samuel","Jones"  
[copy] 2,"Virat","Kumar"  
[copy] 3,"Andrew","Simon"  
[copy] 4,"Raul","A Simpson"  
[copy] \.
```

```
4 rows imported in 1 minute and 24.336 seconds.  
cqlsh:students>
```

**Step 3:** Confirm that the records from the standard input device are loaded into the “persons” table existing in the “students” database.

```
SELECT *  
  FROM persons;
```

```
cqlsh:students> select * from persons;  
 id | fname | lname  
---+-----+-----  
  1 | Samuel | Jones  
  2 | Virat | Kumar  
  4 | Raul | A Simpson  
  3 | Andrew | Simon  
  
(4 rows)  
cqlsh:students>
```

### 3.9.4 Export to STDOUT

***Objective:***

Export the contents of the table/column family “elearninglists” present in the “students” database to the standard output device (STDOUT).

***Act:***

**Step 1:** Check the records of the table “elearninglists” present in the “students” database.

```
SELECT *
  FROM elearninglists;
```

```
cqlsh:students> select * from elearninglists;
  id | course_order | course_id | courseowner | title
-----+-----+-----+-----+-----+
101 |           1 |    1001 | Subhashini | NoSQL Cassandra
101 |           2 |    1002 |      Seema | NoSQL MongoDB
101 |           3 |    1003 |      Seema | Hadoop Sqoop
101 |           4 |    1004 | Subhashini | Hadoop Flume
(4 rows)
```

**Step 2:** Execute the below command at the cqlsh prompt:

```
COPY elearninglists (id, course_order, course_id, courseowner, title) TO
STDOUT;
```

```
cqlsh:students> copy elearninglists (id, course_order, course_id, courseowner, title) to STDOUT;
101,1,1001,Subhashini,NoSQL Cassandra
101,2,1002,Seema,NoSQL MongoDB
101,3,1003,Seema,Hadoop Sqoop
101,4,1004,Subhashini,Hadoop Flume
4 rows exported in 0.031 seconds.
cqlsh:students>
```

## 3.10 QUERYING SYSTEM TABLES

---

The system keyspace includes a number of tables that contain details about your database objects and cluster configuration.

### ***Objective:***

Query the system\_schema.keyspaces table using this SELECT statement.

### ***Act:***

```
SELECT *
  FROM system_schema.keyspaces;
```

## ***Outcome:***

keyspace_name	durable_writes	replication
cycling	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}
test	True	{'Cassandra': '1', 'class': 'org.apache.cassandra.locator.NetworkTopologyStrategy'}
system_auth	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}
system_schema	True	{'class': 'org.apache.cassandra.locator.LocalStrategy'}
dse_system_local	True	{'class': 'org.apache.cassandra.locator.LocalStrategy'}
dse_system	True	{'class': 'org.apache.cassandra.locator.EverywhereStrategy'}
dse_leases	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}
keyspace1	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '2'}
solr_admin	True	{'class': 'org.apache.cassandra.locator.EverywhereStrategy'}
dse_audit	True	{'Cassandra': '2', 'class': 'org.apache.cassandra.locator.NetworkTopologyStrategy'}
system_distributed	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '3'}
system	True	{'class': 'org.apache.cassandra.locator.LocalStrategy'}
dse_perf	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}
system_traces	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '2'}
dse_security	True	{'class': 'org.apache.cassandra.locator.SimpleStrategy', 'replication_factor': '1'}

(15 rows)

---

## ***Objective:***

Get the schema information for tables in the cycling keyspace.

## ***Act:***

```
SELECT *
  FROM system_schema.tables
 WHERE keyspace_name = 'cycling';
```

## ***Outcome:***

```
@ Row 1
+
keyspace_name | cycling
table_name | birthday_list
bloom_filter_fp_chance | 0.01
caching | {'keys': 'ALL', 'rows_per_partition': 'NONE'}
cdc | null
comment |
compaction | {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy', 'max_threshold': 4}
compression | {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
crc_check_chance | 1
dclocal_read_repair_chance | 0.1
default_time_to_live | 0
extensions | {'DSE_RLACA': 0x6379636c6973745f6e616d65}
flags | {'compound'}
gc_grace_seconds | 864000
id | e439b922-2bc5-11e8-891b-23da85222d3d
max_index_interval | 2048
memtable_flush_period_in_ms | 0
min_index_interval | 128
nodesync | null
read_repair_chance | 0
speculative_retry | 99PERCENTILE
...
...
```

### *Objective:*

Get details about a table's columns from the system\_schema.columns table.

### *Act:*

```
SELECT *
  FROM system_schema.columns
 WHERE keyspace_name = 'cycling'
   AND table_name = 'cyclist_name';
```

### *Outcome:*

keyspace_name	table_name	column_name	clustering_order	column_name_bytes	kind	position	type
cycling	cyclist_name	firstname	none	0x66697273746e616d65	regular	-1	text
cycling	cyclist_name	id	none	0x6964	partition_key	0	uuid
cycling	cyclist_name	lastname	none	0x6c6173746e616d65	regular	-1	text

Note: The system\_schema tables do NOT show search index or row-level access control settings.

# REMEMBER ME



1. It is an immensely popular NoSQL owing to its high availability, linear scalability, proven fault tolerance on commodity hardware as well as cloud infrastructure and good performance.

2. Cassandra does NOT have a master-slave architecture, which means it does NOT have a single point of failure.
3. Cassandra employs a peer-to-peer master-less architecture. All the nodes in the cluster are equal. Data is replicated on multiple nodes to ensure fault tolerance and high availability.
4. Gossip protocol is used for intra-ring communication. It is a peer-to-peer communication protocol which eases the discovery and sharing of location and state information with other nodes in the cluster.
5. Cassandra is all for availability. It works on the philosophy that it will always be available for writes.
6. CQL provides a rich set of built-in data types for columns. The counter type is unique. Besides these built-in data types, users can also create their own custom data types.

## TEST ME

---

1. Which database is best suitable for handling data in small volumes?
  - (a) Relational database
  - (b) NoSQL database
  - (c) Both (a) and (b)
  - (d) None of the above
2. What is the full form of CRUD?
  - (a) Create Read Update Delete
  - (b) Change Return Upload Download
  - (c) Create Read Update Drop
  - (d) Both A and C
3. Which of the following is not a part of Cassandra architecture?
  - (a) Column
  - (b) Commit log
  - (c) MemTable
  - (d) Bloom Filter
4. What is the main pre-requisite for Cassandra?
  - (a) CQLSH
  - (b) Memory
  - (c) Java

- (d) Windows OS
5. Which of the following is a shell command in CQLSH?
- (a) Consistency
  - (b) Create Index
  - (c) Insert
  - (d) Color
6. Which of the following is command for verification of the operation?
- (a) Verify
  - (b) Clear
  - (c) Describe
  - (d) Check
7. What command is used to delete all the rows in a table?
- (a) Clear
  - (b) Truncate
  - (c) Delete
  - (d) Drop
8. What command is used to delete the table?
- (a) Clear
  - (b) Truncate
  - (c) Delete
  - (d) Drop
9. Which of the following data type is not used for numerical inputs?
- (a) Variant
  - (b) Float
  - (c) Counter
  - (d) Boolean
10. Which of the following is not a property of Cassandra?
- (a) ACID operations
  - (b) Decentralized deployments
  - (c) Simple transactions
  - (d) Supports every type of data
11. Which of the following decides the distribution of data in a cluster?
- (a) Commit log
  - (b) Gossip protocol

- (c) Compaction
  - (d) Partitioner
12. In which of the following data type is order of the element entered is maintained?
- (a) List
  - (b) Map
  - (c) Set
  - (d) None of the above

## QUESTION ME

---

1. What is Cassandra?
2. Comment on Cassandra writes.
3. What is your understanding of tunable consistency?
4. What are collections in CQLSH? Where are they used?
5. Explain hinted handoffs.
6. What is Cassandra – cli?
7. Explain Cassandra's data model.
8. Explain the replication strategy in Cassandra.
9. Cassandra adheres to the Availability and Partition tolerant traits as stated by the CAP theorem. Explain.

## PRACTICE ME

---

1. ***Objective:***

Create table “elearninglist” with columns: id, course\_order, course\_id, title, courseowner.

***Act:***

```

CREATE TABLE elearninglists (
    id int,
    course_order int,
    course_id int,
    title text,
    courseowner text,
    PRIMARY KEY (id, course_order)
);
Here, id ==> Partition Key, course_order ==> Clustering Column

```

The combination of the id and course\_order in the elearninglists table uniquely identifies a row in the elearninglists table. You can have more than one row with the same id as long as the rows contain different course\_order values.

***Outcome:***

```

cqlsh:students> CREATE TABLE elearninglists (
    ...     id int,
    ...     course_order int,
    ...     course_id int,
    ...     title text,
    ...     courseowner text,
    ...     PRIMARY KEY (id, course_order) );
cqlsh:students>

```

***2. Objective:***

Insert rows into the table “elearninglists”.

***Act:***

```

INSERT INTO elearninglists (id, course_order, course_id, title,
courseowner)
VALUES (101, 1, 1001, 'NoSQL Cassandra', 'Subhashini');

INSERT INTO elearninglists (id, course_order, course_id, title,
courseowner)
VALUES (101, 2, 1002, 'NoSQL MongoDB', 'Seema');

INSERT INTO elearninglists (id, course_order, course_id, title,
courseowner)
VALUES (101, 3, 1003, 'Hadoop Sqoop', 'Seema');

INSERT INTO elearninglists (id, course_order, course_id, title,
courseowner)
VALUES (101, 4, 1004, 'Hadoop Flume', 'Subhashini');

```

***Outcome:***

```
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
...     VALUES (101,1,1001,'NoSQL Cassandra','Subhashini');
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
...     VALUES (101,2,1002,'NoSQL MongoDB','Seema');
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
...     VALUES (101,3,1003,'Hadoop Sqoop','Seema');
Bad Request: line 2:44 no viable alternative at input ')'
cqlsh:students> INSERT INTO elearninglists (id, course_order, course_id, title, courseowner)
...     VALUES (101, 4,1004,'Hadoop Flume', 'Subhashini');
cqlsh:students>
```

***3. Objective:***

Query the table “elearninglists”.

***Act:***

```
SELECT *
  FROM elearninglists;
```

***Outcome:***

```
cqlsh:students> SELECT * FROM elearninglists;
      id | course_order | course_id | courseowner | title
-----+-----+-----+-----+-----+
    101 |            1 |    1001 | Subhashini | NoSQL Cassandra
    101 |            2 |    1002 |      Seema | NoSQL MongoDB
    101 |            4 |    1004 | Subhashini | Hadoop Flume
(3 rows)
```

***4. Objective:***

Query the “elearninglist” table on “courseowner” as a filter.

***Act:***

```
SELECT *
  FROM elearninglists
 WHERE courseowner = 'Seema';
```

***Outcome:***

The query returns an error stating “No indexed columns present in by-columns clause with Equal operator”.

```
cqlsh:students> SELECT * FROM elearninglists WHERE courseowner = 'Seema';
Bad Request: No indexed columns present in by-columns clause with Equal operator
cqlsh:students>
```

### ***Solution:***

Create an index on courseowner column.

```
CREATE INDEX ON
    Elearninglists(courseowner);
```

```
cqlsh:students> CREATE INDEX ON elearninglists(courseowner);
cqlsh:students>
```

Executing the same query now shows the record:

```
cqlsh:students> SELECT * FROM elearninglists WHERE courseowner = 'Seema';
  id | course_order | course_id | courseowner | title
-----+-----+-----+-----+-----+
  101 |          2 |    1002 |      Seema | NoSQL MongoDB
(1 rows)
cqlsh:students>
```

To order all the rows of elearninglists with id = 101 in descending order of “course\_order”, The maximum number of records to retrieve is 50.

```
SELECT *
    FROM elearninglists
      WHERE id = 101
        ORDER BY course_order DESC LIMIT 50;
```

```
cqlsh:students> SELECT * FROM elearninglists WHERE id = 101 ORDER BY course_order DESC LIMIT 50;
  id | course_order | course_id | courseowner | title
-----+-----+-----+-----+-----+
  101 |          4 |    1004 | Subhashini | Hadoop Flume
  101 |          2 |    1002 |      Seema | NoSQL MongoDB
  101 |          1 |    1001 | Subhashini | NoSQL Cassandra
(3 rows)
```

## **REFERENCE ME**

---

**1.**

<http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html>

**2.** <http://www.datastax.com/documentation/cql/3.1/pdf/cql31.pdf>

3. [http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml\\_config\\_consistency\\_c.html](http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html)
4. [http://www.datastax.com/docs/1.0/cluster\\_architecture/about\\_client\\_requests](http://www.datastax.com/docs/1.0/cluster_architecture/about_client_requests)
5. [http://www.datastax.com/docs/datastax\\_enterprise3.1/solutions/about\\_pig](http://www.datastax.com/docs/datastax_enterprise3.1/solutions/about_pig)

## ANSWERS

---

### Try This – 1

1. (b)
2. (b) and (c)

### Test Me

1. (a)
2. (a)
3. (a)
4. (c)
5. (d)
6. (c)
7. (b)
8. (d)
9. (d)
10. (a)
11. (d)
12. (a)



# MongoDB

---

## BRIEF CONTENTS

- What is MongoDB?
- Why MongoDB?
  - Using JSON
- Terms used in RDBMS and MongoDB
  - Document
  - Collection
  - Embedded Document
  - Flexible Schema
  - Run MongoDB
- CRUD Operations
  - Create Database
  - Drop Database
  - Data Types in MongoDB
  - Arrays
  - Aggregate Function
  - MapReduce Framework
  - Cursors in MongoDB

- Indexes
- MongoImport
- MongoExport
- Automatic Generation of Unique Numbers for the “\_id” Field

## 4.1 WHAT IS MongoDB?

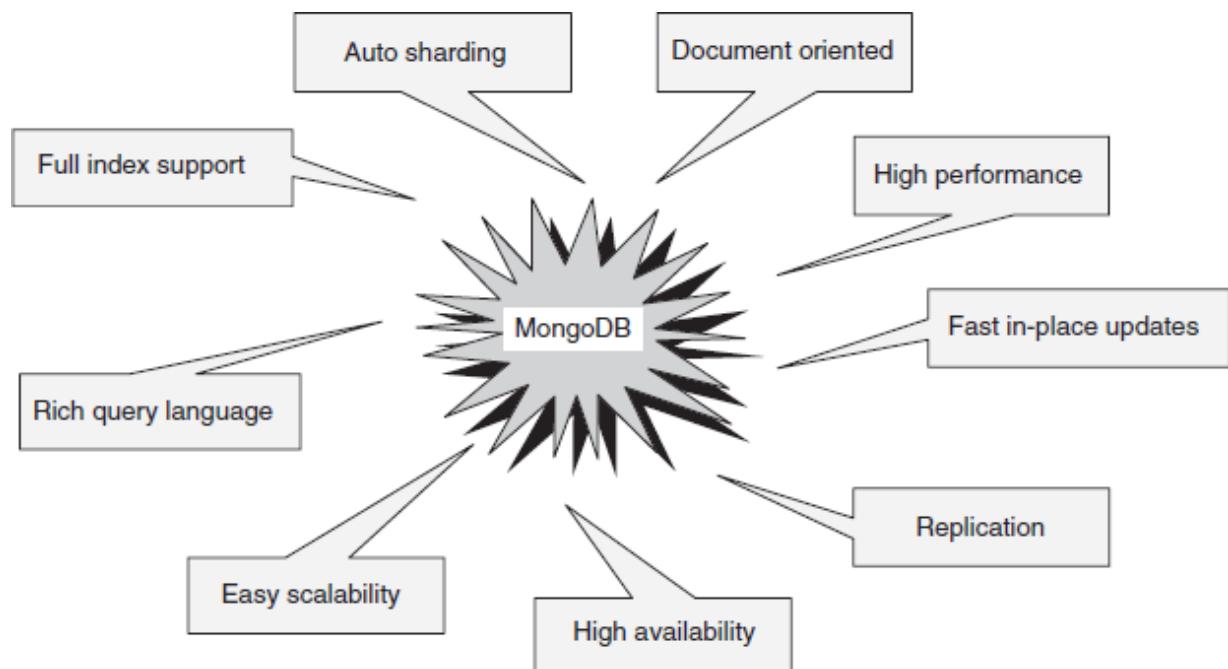
---

The following are features of MongoDB:

1. Cross-platform
2. Open source
3. Non-relational
4. Distributed
5. NoSQL
6. Document-oriented data store

## 4.2 WHY MongoDB?

---



**Figure 4.1** Features of MongoDB.

Figure 4.1 indicates the features of MongoDB which are discussed below:

### Picture This

You are a senior database administrator at the “XYZ” enterprise. You have seen the enterprise grow from few hundred employees to few thousands, and today the company stands tall with over a lakh employee. The data from the various enterprise applications have also grown significantly. You were somehow able to manage the show with a traditional relational database setup. However, of late the requirements have changed and the thought of migrating to a NoSQL database has crossed your mind quite a few times. You have made a wish list (the features that you expect from a NoSQL database).

1. You want a distributed document store that favors Consistency over Availability.
2. You want something relatively easy to set up (*for small clusters*).
3. You plan to use the DB primarily as a Key/Value store with light usage of secondary indexes, Geographic Information System (GIS) or FTS.
4. You want to store some basic time-oriented data like real-time counters.

Enter MongoDB. MongoDB, a NoSQL document database seems the right choice owing to the features listed below:

1. **Document Oriented:** MongoDB is a collection of documents. Each document is a collection of key–value pairs. A single document can be thought of as equivalent of a row and each key as the equivalent of a column. The documents are not required to adhere to a fixed schema. Two documents may or may not have the same attributes.
2. **Easy Scalability:** Let us understand the meaning of a scalable system. A scalable system is one which can maintain or increase its level of performance in the face of increased workloads and great operational demands. One of the characteristic of MongoDB is its easy horizontal scalability, a feat that it achieves with the help of sharding (the word shard means a small part of a whole), distributing data across multiple machines.
3. **Auto Sharding:** What is sharding? Sharding is the process of distributing large data (splitting large datasets into smaller datasets) on multiple server machines for storage. Why is sharding important? With the increase in data size, a single machine may not be able to store all the data and provide an acceptable read and write throughput. MongoDB

sharding helps with horizontal scaling which means as a database administrator you can add more servers to your database. This helps to support data growth by distributing data and load across various servers.

4. **Replication:** What is data replication? Replication is the process of making multiple copies of the data and storing it across multiple databases in multiple locations. Why replication? Replication creates data redundancy which leads to high data availability. Replication helps with disaster recovery (in the event of hardware failure), backup or reporting.
5. **Full Index Support:** Why indexing? Indexing helps with efficient resolution of queries. MongoDB maintains documents in a collection. In the absence of index(s), a search query will lead to scanning each document in the collection. This is considered highly inefficient as it requires the processing of a huge amount of data.
6. **Rich Query Language:** MongoDB has a rich query language with primary and secondary indexes and also supports full text search.

## TRY THIS – 1

---

1. **Who am I?**
  - I am blindingly fast
  - I am massively scalable
  - I am easy to use
  - I work with documents rather than rows
2. **Who am I?**
  - I am not for everyone
  - I am good with complex data structures such as blog posts and comments
  - I am good with analytics such as a real-time Google analytics
  - I am comfortable with Linux, Mac OS, Solaris and Windows
3. **Who am I?**
  - I have support for transactions
  - I have static data
  - I allow dynamic queries to be run on me
4. **Who am I?**

- I am one of the biggest competitor for MongoDB
  - I have dynamic data
  - Only static queries can be run on me
  - I am document-oriented too
- 

### 4.2.1 Using JSON

JSON is extremely expressive. MongoDB actually does not use JSON but BSON (pronounced Bee Son) - it is Binary JSON. It is an open standard. It is used to store complex data structures.

*Let us trace the journey from .csv to XML to JSON.*

Let us look at how data is stored in .csv file. Assume this data is about the employees of an organization named “XYZ”. As can be seen, the column values are separated using commas and the rows are separated by a carriage return.

John, Mathews, +123 4567 8900  
 Andrews, Symmonds, +456 7890 1234  
 Mable, Mathews, +789 1234 5678

This looks good! However, let us make it slightly more legible by adding column heading.

FirstName, LastName, ContactNo  
 John, Mathews, +123 4567 8900  
 Andrews, Symmonds, +456 7890 1234  
 Mable, Mathews, +789 1234 5678

Now assume that few employees have more than one ContactNo. It can be neatly classified as OfficeContactNo and HomeContactNo. But what if few employees have more than one OfficeContactNo and more than one HomeContactNo? This is the first issue we need to address.

Let us look at just another piece of data that you wish to store about the employees. You need to store their email addresses as well. Here again we have the same issues, few employees have two email addresses, few have three and there are a few employees with more than three email addresses as well.

As we come across these fields or columns, we realize that it gets messy with .csv. CSV is known to store data well if it is flat and does not have repeating values.

The problem becomes even more complex when different departments maintain the details of their employees. The formats of .csv (columns, etc.) could vastly differ and it will call for some efforts before we can merge the files from the various departments to make a single file.

This problem can be solved by XML. As the name suggests XML is highly extensible. It does not just call for defining a data format; rather it defines how you define a data format. You may be prepared to undertake this cumbersome task for highly complex and structured data, however for simple data exchange it might just be too much work.

Enter JSON! Let us look at how it reacts to the problem at hand.

```
{  
  FirstName: John,  
  LastName: Mathews,  
  ContactNo: [+123 4567 8900, +123 4444 5555]  
}  
  
{  
  FirstName: Andrews,  
  LastName: Symmonds,  
  ContactNo: [+456 7890 1234, +456 6666 7777]  
}  
  
{  
  FirstName: Mable,  
  LastName: Mathews,  
  ContactNo: +789 1234 5678  
}
```

As you can see it is quite easy to read a JSON. There is absolutely no confusion now. One can have a list of  $n$  contact numbers, and it can be stored with ease.

JSON is very expressive. It provides the much-needed ease to store and retrieve documents in their real form. The binary form of JSON is BSON. BSON is an open standard. In most cases it consumes less space as compared to the text-based JSON. There is yet another advantage with BSON. It is much easier and quicker to convert BSON to a programming language's native data format. There are MongoDB drivers available for a number of programming languages such as C, C++, Ruby, PHP, Python, C#, etc. and each works slightly differently. Use of basic binary format enables the native data structures to be built quickly for each language without going through the hassle of first processing JSON.

## ***Creating or Generating a Unique Key***

Each JSON document should have a unique identifier. It is the `_id` key. It is similar to the primary key in relational databases. This facilitates search for documents based on the unique identifier. An index is automatically built on the unique identifier. It is your choice to either provide unique values yourself or have the Mongo shell generate the same.

0	1	2	3	4	5	6	7	8	9	10	11
Timestamp			Machine ID			Process ID		Counter			

## **Database**

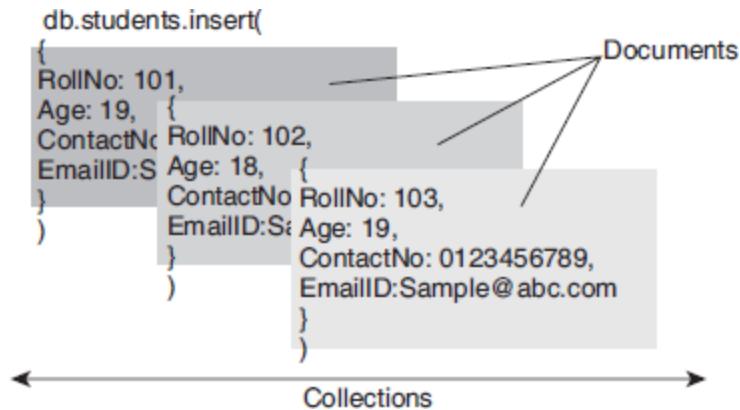
It is a collection of collections. In other words, it is like a container for collections. It gets created the first time that your collection makes a reference to it. This can also be created on demand. Each database gets its own set of files on the file system. A single MongoDB server can house several databases.

## **Collection**

A collection is analogous to a table of RDBMS. A collection is created on demand. It gets created the first time that you attempt to save a document that references it. A collection exists within a single database. It holds several MongoDB documents. A collection does not enforce a schema. This implies that documents within a collection can have different fields. Even if the documents within a collection have same fields, the order of the fields can be different.

## **Document**

A document is analogous to a row/record/tuple in an RDBMS table. A document has a dynamic schema. This implies that a document in a collection need not necessarily have the same set of fields/key-value pairs.



## ***Support for Dynamic Queries***

MongoDB has extensive support for dynamic queries. This is in keeping with traditional RDBMS wherein we have static data and dynamic queries. CouchDB – another document-oriented, schema-less NoSQL database and MongoDB's biggest competitor – works on quiet the reverse philosophy. It has support for dynamic data and static queries.

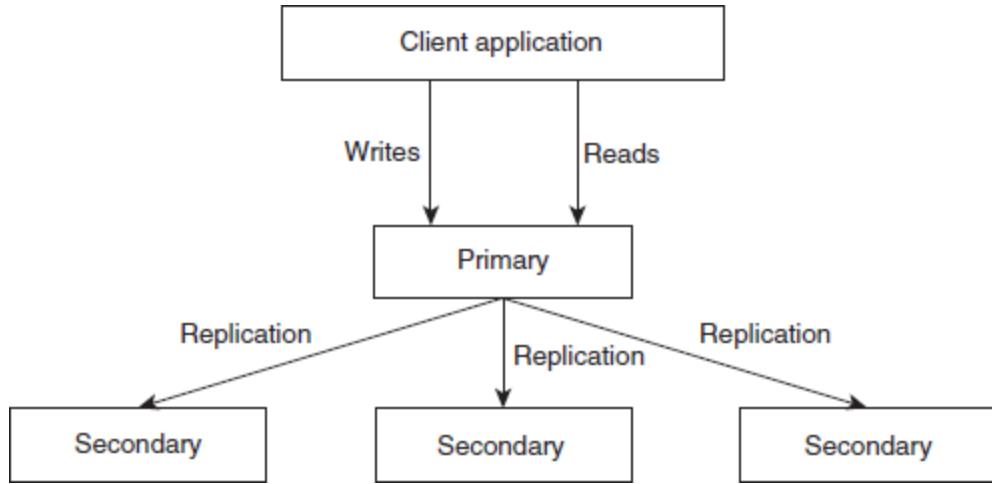
## ***Storing Binary Data***

MongoDB provides GridFS to support the storage of binary data. It can store up to 4 MB of data. This usually suffices for photographs (such as a profile picture) or small audio clips. However, if one wishes to store movie clips, MongoDB has another solution.

It stores the metadata (data about data along with the context information) in a collection called “file”. It then breaks the data into small pieces called chunks and stores it in the “chunks” collection. This process takes care about the need for easy scalability.

## ***Replication***

Why replication? It provides data redundancy and high availability. It helps to recover from hardware failure and service interruptions. In MongoDB, the replica set has a single primary and several secondaries. Each write request from the client is directed to the primary. The primary logs all write requests into its Oplog. The Oplog is then used by the secondary replica members to synchronize their data. This way there is strict adherence to consistency. The clients usually read from the primary. However, the client can also specify a read preference that will then direct the read operations to the secondary.

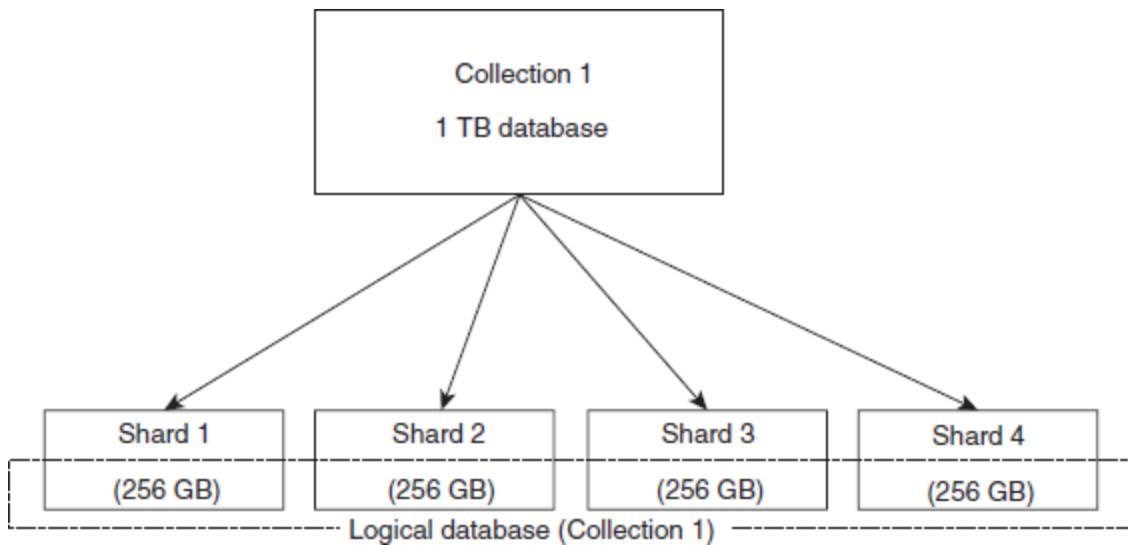


## Sharding

Sharding is akin to horizontal scaling. It means the large dataset is divided and distributed over multiple servers or shards. Each shard is an independent database and collectively they would constitute a logical database.

The prime advantages of sharding are as follows:

1. Sharding reduces the amount of data that each shard needs to store and manage. For example, if the dataset was 1 TB in size and we were to distribute this over four shards, each shard would house just 256 GB data. As the cluster grows, the amount of data that each shard will store and manage will decrease.
2. Sharding reduces the number of operations that each shard handles. For example, if we were to insert data, the application needs to access only that shard which houses that data.



### ***Updating Information In-place***

MongoDB updates the information in-place. This implies that it updates the data wherever it is available. It does not allocate separate space and the indexes remain unaltered.

MongoDB is all for lazy-writes. It writes to disk once every second. Reading and writing to disk is a slow operation as compared to reading and writing from memory. The fewer reads and writes that we perform to the disk, the better is the performance. This makes MongoDB faster than its other competitors who write almost immediately to the disk. However, there is a tradeoff. MongoDB makes no guarantee that data will be stored safely on the disk.

---

## **4.3 TERMS USED IN RDBMS AND MongoDB**

MongoDB is a NoSQL database and it is quite obvious that the terminology used in SQL is not used in MongoDB. However, comparing SQL terms with NoSQL helps to speed up the learning process. A document in MongoDB is equivalent to a record/row in RDBMS/SQL. A group of documents is called *collection in MongoDB*. A collection is similar to the concept of table in RDBMS. Please refer to [Table 4.1](#) for SQL/RDBMS terminology and its counterpart in MongoDB.

**Table 4.1** SQL/RDBMS terminology and its counterpart in MongoDB

RDBMS	MongoDB
Database	Database
Table	Collection
Record	Document or BSON document
Columns	Fields/Key-value pairs
Index	Index
Joins	Embedded documents
Primary key	Primary key ( <code>_id</code> is an identifier)

### 4.3.1 Document

A document is an ordered set of key-value pairs. Each key is unique within a document. It is usually a string value. It is used to retrieve the value. A value is an instance of a supported data type.

**Example:**

Name: “Seema Acharya”

Here, “Name” is the key and the instance of the value is “Seema Acharya”. Since a document is a set of key-value pairs, there is only instance of a member within each document.

**Example:**

```
> db.sample.insert(
... {
... _id:1001,
... Name:"Seema Acharya",
... BookTitle:"Demystifying NoSQL"
... }
...
)
WriteResult({ "nInserted" : 1 })
> db.sample.find().pretty();
{
  "_id" : 1001,
  "Name" : "Seema Acharya",
  "BookTitle" : "Demystifying NoSQL"
}
```

The above is a valid document. Each key, “`_id`”, “Name” and “BookTitle” appears only once within a document.

Let us now look at an invalid document:

```
> db.sample.insert(  
... {  
...   _id:1002,  
...   Name:"Seema Acharya",  
...   Name:"Demystifying NoSQL"  
... }  
... )  
WriteResult({ "nInserted" : 1 })
```

Here the member “Name” appears twice in the document. This is what happens when we retrieve the document. “Name” : “Seema Acharya” is replaced with “Name” : “Demystifying NoSQL”.

```
> db.sample.find().pretty();  
{  
  "_id" : 1001,  
  "Name" : "Seema Acharya",  
  "BookTitle" : "Demystifying NoSQL"  
}  
{ "_id" : 1002, "Name" : "Demystifying NoSQL"}
```

### 4.3.2 Collection

A collection in a document-based data store such as “MongoDB” is a collection of documents.

**Example:**

```

> db.authors.find().pretty();
{
  "_id" : 1001,
  "author" : "Seema Acharya",
  "title" : "Demystifying NoSQL",
  "tags" : [
    "programming",
    "database",
    "mongodb"
  ]
}
{
  "_id" : 1002,
  "author" : "Seema Acharya",
  "title" : "Data analytics using R",
  "tags" : [
    "database",
    "analytics"
  ]
}

```

In the example above, “authors” is the name of the collection and there are two documents (with `_id: 1001` and `_id: 1002`). Besides the `_id` field, the order of the other key–value pairs does not matter. The value for `_id` can be user-defined and if not specified by the user, it will be system-defined. It is like the primary key to uniquely identify a document from the collection.

### 4.3.3 Embedded Document

Assume we have details about the customer and the order that he has placed with an e-commerce site.

Let us look at how it will be stored in a relational system (RDBMS). [Table 4.2](#) stores details about customer and [Table 4.3](#) stores details about the order placed by the customer.

**Table 4.2** Customer table

Column Name	Description
Cust_ID	Customer ID
Cust_Name	Name of the customer
Cust_Type	Type of the customer
Cust_ContactDetails	Customer contact details
....	

**Table 4.3** Order table

Column Name	Description
Cust_ID	Customer ID
Order_ID	Order ID
ProductName	Name of the product
Quantity	Number of units of the product
....	

If one needs to fetch the customer details, along with the details of the order placed by him, a join (which can become a performance bottleneck) will need to be executed on the two tables.

The same can be accomplished in MongoDB using embedded document. This is one of the key differences in how data is modeled in MongoDB.

```
{
  Cust_ID: "C1001",
  Cust_Name: "ABC XYZ",
  Cust_Type: "Premium",
  "Cust_ContactDetails" : 0XXXXXXXXX,
  Order:
  {
    {
      Order_ID: "O101",
      "Product_Name": "P0101",
      Quantity: 5
    }
  }
}
```

#### 4.3.4 Flexible Schema

Think about an e-commerce application. You are a database designer entrusted with the task of designing data models to store the product catalog. There are quite a few products that are sold online. There are mobile phones, camera, books, apparel, etc. Each product type is defined by a set of attributes. For example

A mobile phone is described by: manufacture, front camera, rear camera, internal memory, screen size, price, other specifications etc.

A book is described by: ISBN No., Book title, Author, Publisher, Book type, No. of pages, Year of publication, Price, etc.

In an RDBMS, we may need a separate table for each product type. The columns/fields/attributes to describe a mobile phone are very different from that required to describe a book. Also as we are aware, the schema has to be defined upfront before moving any data into the tables. The number of tables would be as many as the number of product types.

Contrast this to MongoDB. MongoDB offers a flexible schema structure. All the product types can be stored in the same collection. There will be a document for each product type. A document can differ from another document in the number and types of key-value pairs to describe a product type.

### **Examples:**

```
{Product_ID: "P101", ProductType: "Mobile", Manufacturer: "Samsung", FrontCamera: "10 mega pixel", RearCamera: "14 mega pixel", ScreenSize: "3.5 inch", Price: "33000 INR"}
```

```
{Product_ID: "P102", ProductType: "Book", ISBNNo:00123456789123, BookTitle: "Demystifying MongoDB", Author: "Seema Acharya", Publisher: "Wiley India", BookType: "Paper Back", NoofPage: 550, YearofPublication: 2019, Price: "500 INR"}
```

Both the above documents can be placed in the same collection. This is a case of flexible schema.

### **TRY THIS – 2**

---

Which of the following statements are valid with respect to RDBMS and MongoDB?

- (a) MongoDB can store only unstructured data

- (b) In RDBMS, schema needs to be defined before placing data into the database tables
- (c) MongoDB scales out or is horizontally scalable
- (d) A change in the data model/schema does not affect the business logic in RDBMS

### 4.3.5 Run MongoDB

To run MongoDB, follow the three steps listed below:

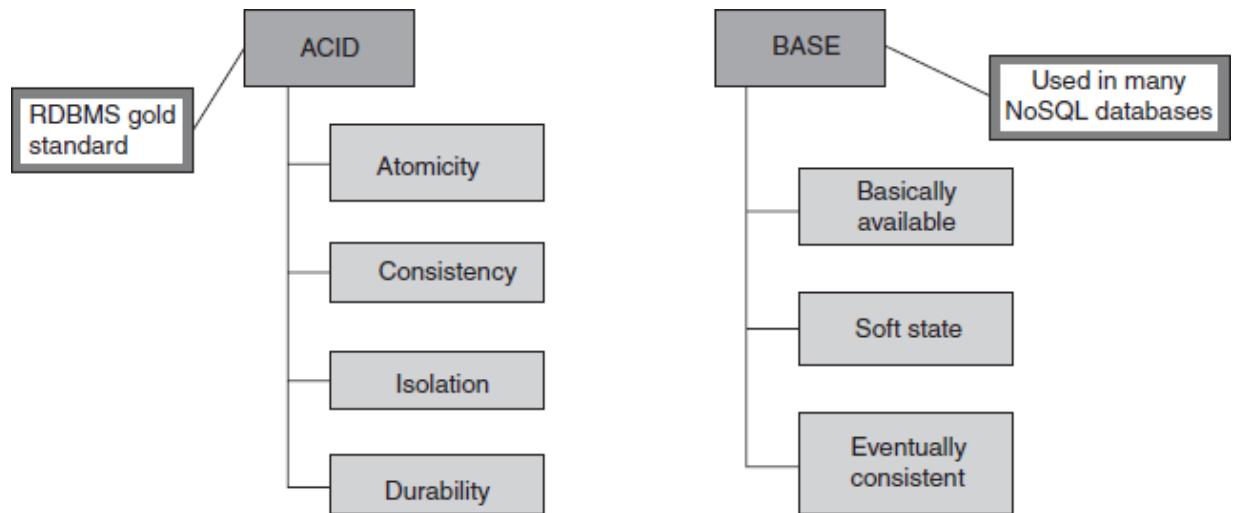
1. Set up the MongoDB environment. MongoDB requires a data directory to store all data. MongoDB's default data directory path is `\data\db`
2. Start MongoDB. To start MongoDB, run **mongod.exe**. Mongod is the "Mongo Daemon". It is basically the host process for the database. Mongod has several default parameters, such as storing data in `/data/db` and running on port 27017.
3. Connect to MongoDB through the `~bin.mongo.exe` shell, open another Command prompt. Mongo is the command-line shell that connects to a specific instance of mongod.

Table 4.4 defines the component set (database server and client) and the binaries in MySQL, Oracle and MongoDB databases.

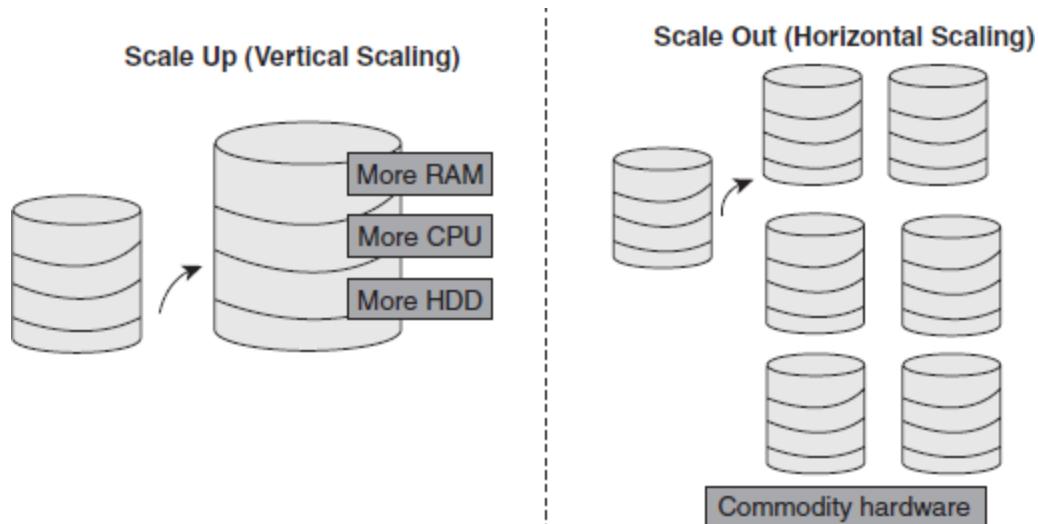
**Table 4.4** Component set and binaries in MySQL, Oracle and MongoDB databases

	<i>MySQL</i>	<i>Oracle</i>	<i>MongoDB</i>
<i>Database Server</i>	MySql	Oracle	Mongod
<i>Database Client</i>	MySql	SQL Plus	Mongo

Apart from the term differences between RDBMS and MongoDB, a few other differences are shown in Figures 4.2 and 4.3.



**Figure 4.2** ACID versus BASE.



**Figure 4.3** Scale-up versus scale-out.

## 4.4 CRUD OPERATIONS

---

### 4.4.1 Create Database

*Syntax:*

```
use DATABASE_Name
```

To create a database by the name “myDB” use the following command:

```
use myDB
```

```
> use myDB;  
switched to db myDB  
>
```

To confirm the existence of your database, type the following command at the MongoDB shell:

```
db
```

```
> db;  
myDB  
>
```

To get a list of all databases, type the command

```
show dbs
```

```
> show dbs:  
admin      (empty)  
local      0.078GB  
test       0.078GB  
>
```

Notice that the newly created database “myDB” does not show up in the list above. The reason is that the database needs to have at least one document to show up in the list.

The default database in MongoDB is test. If one does not create any database, then all collections are by default stored in the test database.

#### 4.4.2 Drop Database

*Syntax:*

```
db.dropDatabase();
```

To drop the database “myDB”, first ensure that you are currently placed in “myDB” database and then use the db.dropDatabase() command to drop the

database.

```
use myDB;  
db.dropDatabase();
```

Confirm if the database “myDB” has been dropped.

```
> db.dropDatabase();  
{ "dropped" : "myDB", "ok" : 1 }  
>
```

If no database is selected, the default database “test” is dropped.

#### 4.4.3 Data Types in MongoDB

Table 4.5 provides details of the different data types in MongoDB.

**Table 4.5** Data types in MongoDB

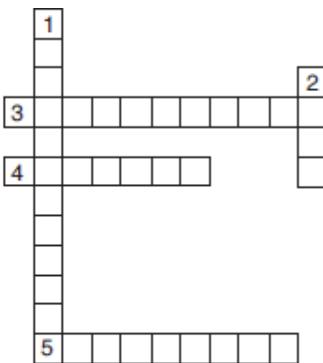
Data type	Description
String	Must be UTF-8 valid Most commonly used data type
Integer	Can be 32-bit or 64-bit (depends on the server)
Boolean	To store a true/false value
Double	To store floating point (real values)
Min/Max keys	To compare a value against the lowest or highest BSON elements
Arrays	To store arrays or list or multiple values into one key
Timestamp	To record when a document has been modified or added
Null	To store a NULL value. A NULL is a missing or unknown value
Date	To store the current date or time in Unix time format. One can create object of date and pass day, month and year to it.
Object ID	To store the document's id
Binary data	To store binary data (images, binaries, etc.)
Code	To store JavaScript code into the document
Regular expression	To store regular expression

## TRY THIS – 3

Solve this basic puzzle on MongoDB.

**Across**

- 3 MongoDB database stores its data in -----
- 4 MongoDB uses ----- schemas.
- 5 A collection holds one or more -----



**Down**

- 1 MongoDB uses ----- files.
- 2 MongoDB uses -----, a binary object format similar to, but more expensive than JSON.

Trying out a few commands:

1. To report the name of the current database.

```
C:\Windows\system32\cmd.exe - mongo
> db
test
>
```

2. To display the list of databases.

```
C:\Windows\system32\cmd.exe - mongo
> show dbs
admin (empty)
local 0.078GB
myDB1 0.078GB
>
```

3. To switch to a new database, for example, myDB1.

```
C:\windows\system32\cmd.exe - mongo
> use myDB1
switched to db myDB1
>
```

4. To display the list of collections (tables) in the current database.

```
C:\windows\system32\cmd.exe - mongo
> show collections
system.indexes
system.js
>
```

5. To display the current version of the MongoDB server.

```
C:\windows\system32\cmd.exe - mongo
> db.version()
2.6.1
>
```

6. To display the statistics that reflects the use state of a database.

```
C:\windows\system32\cmd.exe - mongo
> db.stats()
{
  "db" : "myDB1",
  "collections" : 3,
  "objects" : 6,
  "avgObjSize" : 122.66666666666667,
  "dataSize" : 736,
  "storageSize" : 24576,
  "numExtents" : 3,
  "indexes" : 1,
  "indexSize" : 8176,
  "fileSize" : 67108864,
  "nsSizeMB" : 16,
  "dataFileVersion" : {
    "major" : 4,
    "minor" : 5
  },
  "extentFreeList" : {
    "num" : 14,
    "totalSize" : 974848
  },
  "ok" : 1
}
```

Type db.help() in the MongoDB client to get a list of commands:

```
C:\Windows\system32\cmd.exe - mongo
> db.help();
DB methods:
  db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command
  [ just calls db.runCommand(...) ]
  db.auth(username, password)
  db.cloneDatabase(fromhost)
  db.commandHelp(name) returns the help for the command
  db.copyDatabase(fromdb, todb, fromhost)
  db.createCollection(name, { size : ..., capped : ..., max : ... } )
  db.createUser(userDocument)
  db.currentOp() displays currently executing operations in the db
  db.dropDatabase()
  db.eval(func, args) run code server-side
  db.fsyncLock() flush data to disk and lock server for backups
  db.fsyncUnlock() unlocks server following a db.fsyncLock()
  db.getCollection(cname) same as db['cname'] or db.cname
  db.getCollectionNames()
  db.getLastErr() - just returns the err msg string
  db.getLastErrObj() - return full status object
  db.getMongo() get the server connection object
  db.getMongo().setSlaveOk() allow queries on a replication slave server
  db.getName()
  db.getPrevError()
  db.getProfilingLevel() - deprecated
  db.getProfilingStatus() - returns if profiling is on and slow threshold
  db.getReplicationInfo()
  db.getSiblingDB(name) get the db at the same server as this one
  db.getWriteConcern() - returns the write concern used for any operations
  on this db, inherited from server object if set
  db.hostInfo() get details about the server's host
  db.isMaster() check replica primary status
  db.killOp(copid) kills the current operation in the db
  db.listCommands() lists all the db commands
  db.loadServerScripts() loads all the scripts in db.system.js
  db.logout()
  db.printCollectionStats()
  db.printReplicationInfo()
  db.printShardingStatus()
  db.printSlaveReplicationInfo()
  db.dropUser(username)
  db.repairDatabase()
  db.resetError()
  db.runCommand(cmdObj) run a database command. if cmdObj is a string,
  turns it into { cmdObj : 1 }
  db.serverStatus()
  db.setProfilingLevel(level,<slowms>) 0=off 1=slow 2=all
  db.setWriteConcern( <write concern doc> ) - sets the write concern for
  writes to the db
  db.unsetWriteConcern( <write concern doc> ) - unsets the write concern
  for writes to the db
  db.setVerboseShell(flag) display extra information in shell output
  db.shutdownServer()
  db.stats()
  db.version() current version of the server
>
```

Let us create a user “Seema” with the login credentials (password set to “gurukul”).

```
db.createUser({ "user": "Seema", "pwd": "gurukul", "roles": [ "readWrite",
  "dbAdmin" ] });
```

```

> db.createUser({ "user": "Seema", "pwd": "gurukul", "roles": [ "readwrite", "dbAdmin" ] });
Successfully added user: { "user": "Seema", "roles": [ "readwrite", "dbAdmin" ] }
> show users;
{
  "_id" : "test.Seema",
  "user" : "Seema",
  "db" : "test",
  "roles" : [
    {
      "role" : "readwrite",
      "db" : "test"
    },
    {
      "role" : "dbAdmin",
      "db" : "test"
    }
  ]
}

```

To login into mongo shell use the following command:

```

D:\BackupfromOldLaptop\2015\MongoDB\bin>mongo -u Seema -p gurukul --authenticationDatabase test
MongoDB shell version: 2.6.1
connecting to: test
Server has startup warnings:
2019-03-03T13:33:49.766+0530 [initandlisten]
2019-03-03T13:33:49.767+0530 [initandlisten] ** NOTE: This is a 32 bit MongoDB binary.
2019-03-03T13:33:49.768+0530 [initandlisten] ** 32 bit builds are limited to less than 2GB of data (or less with
--journal).
2019-03-03T13:33:49.769+0530 [initandlisten] ** Note that journaling defaults to off for 32 bit and is currently
off.
2019-03-03T13:33:49.771+0530 [initandlisten] ** See http://dochub.mongodb.org/core/32bit
2019-03-03T13:33:49.771+0530 [initandlisten]

```

Consider a table “Students” with the following columns:

1. StudRollNo
2. StudName
3. Grade
4. Hobbies
5. DOJ

Before we get into the details of CRUD operations in MongoDB, let us look at how the statements are written in RDBMS and MongoDB ([Table 4.6](#)).

**Table 4.6** Statements written in RDBMS and MongoDB

	RDBMS	MongoDB
<b>Insert</b>	Insert into Students (StudRollNo, StudName, Grade, Hobbies, DOJ) Values ('S101', 'Simon David', 'VII', 'Net Surfing', '10-Oct-2012')	db.Students.insert({_id:1, StudRollNo: 'S101', StudName: 'Simon David', Grade: 'VII', Hobbies: 'Net Surfing', DOJ: '10-Oct-2012'});
<b>Update</b>	Update Students set Hobbies = 'Ice Hockey' where StudRollNo = 'S101'  Update Students Set Hobbies = 'Ice Hockey'	db.Students.update({StudRollNo: 'S101'}, {\$set : {Hobbies : 'Ice Hockey'}})  db.Students.update({}, {\$set: {Hobbies: 'Ice Hockey' }}, {multi:true})
<b>Delete</b>	Delete from Students where StudRollNo = 'S101'  Delete From Students	db.Students.remove ({StudRollNo : 'S101'})  db.Students.remove({})
<b>Select</b>	Select * from Students  Select * from students where StudRollNo = 'S101'	db.Students.find() db.Students.find().pretty()  db.Students.find({StudRollNo: 'S101'})
	Select StudRollNo, StudName, Hobbies from Students	db.Students.find({}, {StudRollNo:1, StudName:1, Hobbies:1, _id:0})
	Select StudRollNo, StudName, Hobbies from Students where StudRollNo = 'S101'	db.Students.find({StudRollNo: 'S101'}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
	Select StudRollNo, StudName, Hobbies From Students Where Grade = 'VII' and Hobbies = 'Ice Hockey'	db.Students.find({Grade: 'VII' , Hobbies: 'Ice Hockey'}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
	Select StudRollNo, StudName, Hobbies From Students Where Grade = 'VII' or Hobbies = 'Ice Hockey'	db.Students.find({ \$or: [{Grade: 'VII' , Hobbies: 'Ice Hockey'}]}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
	Select * From Students Where StudName like 'S%'	db.Students.find({ StudName: /^S/}).pretty()

---

**Objective:** To create a collection by the name “Person”.

Let us take a look at the collection list prior to the creation of the new collection “Person”:

```
C:\windows\system32\cmd.exe - mongo
> show collections
Students
food
system.indexes
system.js
>
```

**Act:** The statement:

```
db.createCollection("Person")
```

```
C:\windows\system32\cmd.exe - mongo
> db.createCollection("Person");
{ "ok" : 1 }
>
```

**Outcome:** Below is the collection list after the creation of the new collection “Person”:

```
C:\windows\system32\cmd.exe - mongo
> show collections;
Person
Students
food
system.indexes
system.js
>
```

---

**Objective:** To drop a collection by the name “food”.

Take a look at the current collection list:

```
C:\windows\system32\cmd.exe - mongo
> show collections;
Person
Students
food
system.indexes
system.js
>
```

**Act:** The statement to drop the collection:

```
db.food.drop();
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.drop();
true
>
```

The drop() method will return true if the selected collection was dropped successfully, otherwise it will return false.

**Outcome:** The collection list after the execution of the statement:

```
C:\windows\system32\cmd.exe - mongo
> show collections
Person
Students
system.indexes
system.js
>
```

---

Let us look at some of the options (such as “capped”, “autoIndexId”, “size” and “max”) available with db.createCollection() method.

```
> db.createCollection("Person", { capped : true, autoIndexId : true, size : 6142800, max : 10000 } );
{ "ok" : 1 }
> show collections
Person
system.indexes
>
```

- 1. Capped:** It accepts a Boolean value. If the value is true, it enables a capped collection. If capped is specified as true, it is mandatory to specify the size as well. Capped means there will be a limit on the size of the collection. It will automatically start to overwrite the older entries when it reaches the maximum size.

2. **AutoIndexID:** This parameter too accepts a Boolean value. By default this is false. When true, it will automatically create an index on the `_id` field.
3. **Size:** It accepts a number value. It is compulsory to provide a value for size in bytes if capped is set to true.
4. **Max:** It accepts a number value. This is used to specify the maximum number of documents allowed in a capped collection.

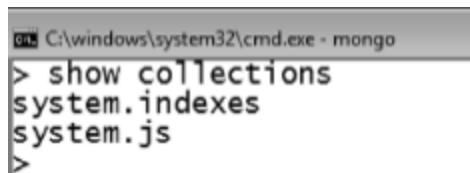
## TRY THIS – 4

A collection “Person” has been created with capped set to true, size as 80 bytes and a maximum of two documents. What will be the output of “db.Person.find();”?

```
> db.Person.drop();
true
> db.createCollection("Person", { capped : true, autoIndexId : true, size : 80, max : 2 });
{ "ok" : 1 }
> db.Person.insert({ _id:1, Name: "Seema Acharya"});
WriteResult({ "nInserted" : 1 })
> db.Person.insert({ _id:2, Name: "Sunita Acharya"});
WriteResult({ "nInserted" : 1 })
> db.Person.insert({ _id:3, Name: "Jeet Acharya"});
WriteResult({ "nInserted" : 1 })
> db.Person.insert({ _id:4, Name: "Sanjeet Acharya"});
WriteResult({ "nInserted" : 1 })
> db.Person.find()
```

**Objective:** To create a collection by the name “Students” and insert documents.

**Input:** Check the list of existing collections.



```
C:\windows\system32\cmd.exe - mongo
> show collections
system.indexes
system.js
>
```

Before using the insert method, let us understand its syntax.

Syntax of insert method:

```

db.students.insert( ← Collection
{
  RollNo: 101, ← Field: Value
  Age: 19, ← Field: Value
  ContactNo: 0123456789, ← Field: Value
  EmailID:Sample@abc.com ← Field: Value
}
)

```

In the above, “students” is the name of the collection, insert is the method, RollNo, Age, ContactNo and EmailID are the fields. “101”, “19”, “0123456789”, and Sample@abc.com are the values for fields, RollNo, Age, ContactNo, and EmailID.

**Act:** Create a collection by the name “Students” and store the following data in it:

StudName = “Michelle Jacintha”,  
 Grade = “VII”  
 Hobbies = “Internet Surfing”

**Command:**

```
db.Students.insert({_id:1, StudName:"Michelle Jacintha", Grade: "VII", Hobbies:
  "Internet Surfing"});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.insert({_id:1, StudName:"Michelle Jacintha", Grade: "VII", Hobbies: "Internet Surfing"});
> writeResult({ "nInserted" : 1 })
```

**Outcome:** Check if the collection has been successfully created.

```
C:\windows\system32\cmd.exe - mongo
> show collections
Students
system.indexes
system.js
>
```

In place of “show collections”, one can also use “show tables”.

```
> show tables
Students
system.indexes
>
```

One can also use the method `getCollectionNames()` to get a list of collections.

```
> db.getCollectionNames();
[ "Students", "system.indexes" ]
>
```

Check if the document for Student “Michelle Jacintha” has been successfully inserted into the “Students” collection.

```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find();
{ "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII", "Hobbies" : "Internet Surfing" }
>
```

To format the result, one can add the `pretty()` method to the `find` method.

```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
>
```

---

**Objective:** Insert another document into the collection.

**Input:** Check the documents in the “Students” collection before proceeding.

```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
>
```

**Act:**

```
db.Students.insert({_id:2, StudName:"Mabel Mathews", Grade: "VII", Hobbies:
"Baseball"});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.insert({_id:2, StudName:"Mabel Mathews", Grade: "VII", Hobbies: "Baseball"});
WriteResult({ "nInserted" : 1 })
>
```

### **Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
>
```

---

**Objective:** Find the document in the “Students” collection wherein the Grade is “VII” and Hobbies is set to “Baseball”.

### **Act:**

```
db.Students.find( {
  $and: [
    {Grade: "VII"}, {Hobbies: "Baseball"}
  ] } ).pretty();
```

### **Outcome:**

```
> db.Students.find( { $and: [ {Grade: "VII"}, {Hobbies: "Baseball"} ] } ).pretty();
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
>
```

---

**Objective:** Insert the document for “Aryan David” into the Students collection, only if it does not already exist in the collection. However, if it is already present in the collection, then update the document with new values.

(Update his Hobbies from “Skating” to “Chess”.) Use “**Update else insert**” (if there is an existing document, it will attempt to update it; if there is no existing document then it will insert it).

**Input:** Check the documents in the “Students” collection before proceeding.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
>
```

**Act:**

```
db.Students.update({_id:3, StudName:"Aryan David", Grade: "VII"}, {$set:{Hobbies: "Skating"}}, {upsert:true});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:3, StudName:"Aryan David", Grade: "VII"}, {$set:{Hobbies: "Skating"}}, {upsert:true});
writeResult({ "nMatched" : 0, "nUpserted" : 1, "nModified" : 0, "_id" : 3 })
>
```

**Outcome:** Confirm the presence of the document of “Aryan David” in the “Students” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:3});
{ "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Skating" }
>
```

---

**Objective:** Insert the document for “Aryan David” into the Students collection only if it does not already exist in the collection. However, if it is already present in the collection, then update the document with new values. (Update his Hobbies from “Skating” to “Chess”.) Use “**Update else insert**” (if there is an existing document, it will attempt to update it; if there is no existing document then it will insert it).

**Input:** Check the documents in the “Students” collection before proceeding.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Skating"
}
```

**Act:**

```
db.Students.update({_id:3, StudName:"Aryan David", Grade: "VII"}, {$set:{Hobbies: "Chess"}}, {upsert:true});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:3, StudName:"Aryan David", Grade: "VII"}, {$set:{Hobbies: "Chess"}}, {upsert:true});
> writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

**Outcome:** Confirm that the required changes have been made to the document of “Aryan David” in the “Students” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:3});
{ "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }
```

---

**Objective:** To demonstrate Save method to insert a document for student “Vamsi Bapat” in the “Students” collection. Omit providing value for the \_id key.

**Input:** Check the documents in the “Students” collection before proceeding.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
>
```

*Act:*

```
db.Students.save({StudName:"Vamsi Bapat",Grade:"VI"})
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.save({StudName:"Vamsi Bapat",Grade:"VI"})
writeResult({ "nInserted" : 1 })
>
```

**Outcome:** Confirm the presence of the document of “Vamsi Bapat” in the “Students” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}

{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}

{
  "_id" : ObjectId("546dd0e0a7fba710799bb94d"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
```

*Explanation of save() method:* The save() method will insert a new document if the document with the specified \_id does not exist. However, if a document with the specified \_id exists, it replaces the existing document with the new one.

**Objective:** Insert the document of “Hersch Gibbs” into the Students collection using the Update method. First try with upsert set to false and then with upsert set to true.

**Input:** Check the documents in the “Students” collection before proceeding.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find();
{
  "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII", "Hobbies" : "Internet Surfing" }
{
  "_id" : 2, "StudName" : "Mabel Mathews", "Grade" : "VII", "Hobbies" : "Baseball" }
{
  "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }
{
  "_id" : ObjectId("546dd0e0a7fba710799bb94d"), "StudName" : "Vamsi Bapat", "Grade" : "VI" }
```

**Act:** Update method with upsert set to false.

```
db.Students.update({_id:4, StudName:"Hersch Gibbs", Grade:"VII"}, {$set:{Hobbies: "Graffiti"}}, {upsert:false});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:4, StudName:"Hersch Gibbs", Grade:"VII"}, {$set:{Hobbies: "Graffiti"}}, {upsert:false});
writeResult({ "nMatched" : 0, "nUpserted" : 0, "nModified" : 0 })
```

As evident from the above display (nUpserted : 0) , no document has been inserted because upsert is set to false.

Update method with upsert set to true.

```
db.Students.update({_id:4, StudName:"Hersch Gibbs", Grade: "VII"}, {$set:{Hobbies: "Graffiti"}}, {upsert:true});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:4, StudName:"Hersch Gibbs", Grade: "VII"}, {$set:{Hobbies: "Graffiti"}}, {upsert:true});
> writeResult({ "nMatched" : 0, "nUpserted" : 1, "nModified" : 0, "_id" : 4 })
>
```

nUpserted: 1, implying that a document with \_id:4 has been inserted.

**Outcome:** Confirm the presence of the document of “Hersch Gibbs” in the “Students” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find();
> {"_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII", "Hobbies" : "Internet Surfing" }
> {"_id" : 2, "StudName" : "Mabel Mathews", "Grade" : "VII", "Hobbies" : "Baseball" }
> {"_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }
> {"_id" : ObjectId("546dd0e0a7fba710799bb94d"), "StudName" : "Vamsi Bapat", "Grade" : "VI" }
> {"_id" : 4, "Grade" : "VII", "StudName" : "Hersch Gibbs", "Hobbies" : "Graffiti" }
```

---

**Objective:** To insert a single document into a “Customers” collection using “insertOne” method.

**Act:**

```
db.Customers.insertOne({_id:2, CustName: "Mabel Mathews", Hobbies: "Baseball"});
```

```
> db.Customers.insertOne({_id:2, CustName:"Mabel Mathews", Hobbies:"Baseball"});
{ "acknowledged" : true, "insertedId" : 2 }
> db.Customers.find().pretty();
{ "_id" : 2, "CustName" : "Mabel Mathews", "Hobbies" : "Baseball" }
>
```

---

**Objective:** To insert three documents into the “Customers” collection using “insertMany” method.

**Act:**

```
> db.Customers.insertMany(  
... [  
... {_id:11, CustName:"Anaitha G", Hobbies:"Baseball", lvl:2},  
... {_id:12, CustName:"Samanth P", Hobbies:"Soccer", lvl:2},  
... {_id:13, CustName:"Nick Jonathan", Hobbies:"Skating", lvl:2}  
... ]);  
{ "acknowledged" : true, "insertedIds" : [ 11, 12, 13 ] }  
>
```

### ***Outcome:***

```
> db.Customers.find().pretty();  
{ "_id" : 2, "CustName" : "Mabel Mathews", "Hobbies" : "Baseball" }  
{ "_id" : 11, "CustName" : "Anaitha G", "Hobbies" : "Baseball", "lvl" : 2 }  
{ "_id" : 12, "CustName" : "Samanth P", "Hobbies" : "Soccer", "lvl" : 2 }  
{  
    "_id" : 13,  
    "CustName" : "Nick Jonathan",  
    "Hobbies" : "Skating",  
    "lvl" : 2  
}  
>
```

---

***Objective:*** Perform the following tasks on the “Customers” collection using the “bulkWrite” method:

- (a) Delete the document with the value of 12 in the `_id` column.
- (b) Insert a document with `_id` value of 2. This should result in an error as there is already a document with an `_id` value of 2.
- (c) Update the document with `_id` value of 2. The “Hobbies” key should have its value updated to “Basketball” from “Baseball”.
- (d) Replace the document with `_id` value of “2” with a new document.

### ***Act:***

```

> db.Customers.bulkWrite([
... {deleteOne: {"filter": {"_id":12}}},
... {insertOne:{document:
... {_id:2, CustName:"Mabel Mathews", Hobbies:"Baseball"}},
... {updateOne:
... "filter": {"_id":2},
... "update":{$set:{Hobbies:"Basketball"}}
... }},
... {replaceOne:
... "filter": {"_id":2},
... "replacement": {"_id":2,"CustName": "Mapel Mathews","Hobbies": "Ice Skating"}
... }]);
2019-03-06T16:14:11.207+0530 E QUERY    [js] BulkWriteError: write error at item 1 in bulk operation :
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: test.Customers index: _id_ dup key: { : 2.0 }",
            "op" : {
                "_id" : 2,
                "CustName" : "Mabel Mathews",
                "Hobbies" : "Baseball"
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 0,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 1,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:369:48
BulkWriteResult>this.toError@src/mongo/shell/bulk_api.js:333:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1173:1
DBCollection.prototype.bulkWrite@src/mongo/shell/crud_api.js:201:20
@(shell):1:1

```

**Outcome:** The “deleteOne” statement was successfully executed. The document with “\_id” value of “12” was successfully removed from the “Customers” collection.

The “insertOne” statement failed as a document with “\_id” value of “2” already exists in the collection.

No other statements, “updateOne” and “replaceOne”, could be performed as the “insertOne” statement could not be successfully executed.

```

> db.Customers.find().pretty();
{
  "_id" : 2, "CustName" : "Mabel Mathews", "Hobbies" : "Baseball" }
{
  "_id" : 11, "CustName" : "Anaitha G", "Hobbies" : "Baseball", "lvl" : 2 }
{
  "_id" : 13,
  "CustName" : "Nick Jonathan",
  "Hobbies" : "Skating",
  "lvl" : 2
}
>

```

What should we do in order to execute all statements written after a statement that was not successfully executed? In the example above, we would like the updateOne and replaceOne to execute despite the failure of the “insertOne” statement. The same can be achieved with the bulkWrite method with “ordered” set to “false”.

```
> db.Customers.bulkWrite([
... {deleteOne: {"filter": {"_id":12}}},
... {insertOne:{"document":
... {_id:2, CustName:"Mabel Mathews", Hobbies:"Baseball"}}, 
... {updateOne:{ 
... "filter": {"_id":2}, 
... "update":{$set:{ "Hobbies": "Basketball"} } 
... }},
... {replaceOne:{ 
... "filter": {"_id":2}, 
... "replacement": {"_id":2,"CustName": "Mapel Mathews","Hobbies": "Ice Skating"} 
... }}, {ordered:false}]);
2019-03-06T16:19:10.631+0530 E QUERY    [js] BulkWriteError: write error at item 1 in bulk operation :
BulkWriteError({
  "writeErrors" : [
    {
      "index" : 1,
      "code" : 11000,
      "errmsg" : "E11000 duplicate key error collection: test.Customers index: _id_ dup key: { : 2.0 }",
      "op" : {
        "_id" : 2,
        "CustName" : "Mabel Mathews",
        "Hobbies" : "Baseball"
      }
    }
  ],
  "writeConcernErrors" : [ ],
  "nInserted" : 0,
  "nUpserted" : 0,
  "nMatched" : 1,
  "nModified" : 1,
  "nRemoved" : 0,
  "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:369:48
BulkWriteResult>this.toError@src/mongo/shell/bulk_api.js:333:24
Bulk>this.execute@src/mongo/shell/bulk_api.js:1173:1
DBCollection.prototype.bulkWrite@src/mongo/shell/crud_api.js:201:20
@(shell):1:1
```

### ***Outcome:***

```
> db.Customers.find().pretty();
{ "_id" : 2, "CustName" : "Mapel Mathews", "Hobbies" : "Ice Skating" }
{ "_id" : 11, "CustName" : "Anaitha G", "Hobbies" : "Baseball", "lvl" : 2 }
{
  "_id" : 13,
  "CustName" : "Nick Jonathan",
  "Hobbies" : "Skating",
  "lvl" : 2
}
>
```

---

**Objective:** Delete the first document with the “lvl” column set to “2”.

**Act:**

```
db.Customers.deleteOne({"lvl":2});
```

```
> db.Customers.deleteOne({"lvl":2});
{ "acknowledged" : true, "deletedCount" : 1 }
```

**Outcome:**

```
> db.Customers.find().pretty();
{ "_id" : 2, "CustName" : "Mabel Mathews", "Hobbies" : "Baseball" }
{
    "_id" : 13,
    "CustName" : "Nick Jonathan",
    "Hobbies" : "Skating",
    "lvl" : 2
}
>
```

---

**Objective:** Delete all the documents with “lvl” column set to “2”.

**Act:**

```
db.Customers.deleteMany({"lvl":2});
```

```
> db.Customers.deleteMany({"lvl":2});
{ "acknowledged" : true, "deletedCount" : 1 }
> db.Customers.find().pretty();
{ "_id" : 2, "CustName" : "Mabel Mathews", "Hobbies" : "Baseball" }
>
```

## TRY THIS – 5

---

1. Which of the following representation of documents are valid?  
(a) { \_id: 1111, sname: “Aman”, deptno: 10, sub: (CSc, Security, Networking) }

- (b) { \_id: 1111, sname: "Aman", deptno: 10 }
- (c) { sname: "Aman", deptno: 10, sub: ["CSc", "Security", "Networking"] }
2. Select the MongoDB equivalent for the below SQL query:
- ```
INSERT INTO employee (eid, age, salary) VALUES (763357, 45, 56489.40);
```
- (a) db.employee.insert( eid:763357, age: 45, salary: 56489.40 )
- (b) db.employee.insert{ ( eid:763357, age: 45, salary: 56489.40 ) }
- (c) db.employee.insert( { eid:763357, age: 45, salary: 56489.40 } )
- (d) All the options are valid
3. Identify the invalid statement/s.
- (a) db.employee.insertMany( { [ \_id:1,ename:"AAA" ] } )
- (b) db.employee.insertMany( [ { \_id:1,ename:"AAA" } ] )
- (c) db.employee.insertMany( [ { \_id:1,ename:"AAA" }, { \_id:2,ename:"BBB" } ] )
- (d) db.employee.insert( [ \_id:4,ename:"AAA" ] )

## To Add a New Field to an Existing Document

Explaining the syntax of update method:

```
db.students.update( ← Collection
  {Age: {$gt 18}}, ← Update Criteria
  {$set: {Status: "A" }}, ← Update Action
  {multi:true}, ← Update Option
)
```

**Objective:** To add a new field "Location" with value "Newark" to the document (\_id:4) of "Students" collection.

**Input:** Check the document (\_id:4) in the "Students" collection before proceeding.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:4}).pretty();
{
    "_id" : 4,
    "Grade" : "VII",
    "StudName" : "Hersch Gibbs",
    "Hobbies" : "Graffiti"
}
>
```

*Act:*

```
db.Students.update({_id:4}, {$set:{Location:"Newark"} }) ;
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:4}, {$set:{Location:"Newark"} });
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** Confirm that the new field “Location” with value “Newark” has been added to document (\_id:4) in the “Students” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:4}).pretty();
{
    "_id" : 4,
    "Grade" : "VII",
    "StudName" : "Hersch Gibbs",
    "Hobbies" : "Graffiti",
    "Location" : "Newark"
}
>
```

---

## To Remove an Existing Field from an Existing Document

Explaining the syntax of remove method:

```
db.students.remove(           ← Collection
                    (Age: {$gt 18}), ← Remove Criteria
                    )
```

**Objective:** To remove the field “Location” with value “Newark” in the document (\_id:4) of “Students” collection.

**Input:** Check the document (\_id:4) in the “Students” collection before proceeding.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:4}).pretty();
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti",
  "Location" : "Newark"
}
>
```

**Act:**

```
db.Students.update({_id:4}, {$unset:{Location:"Newark"}});

```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:4}, {$unset:{Location:"Newark"}});
writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** Confirm if the stated field (“Location”) has been dropped from the document (\_id:4) of the “Students” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:4}).pretty();
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
>
```

---

## Finding Documents Based on Search Criteria

Explaining the syntax of find method:

```

db.students.find(      ← Collection
  {Age: {$gt 18}},    ← Selection Criteria
  {RollNo:1, Age:1, _id:1} ← Projection
  ).limit(10)         ← Cursor Modifier

```

**Objective:** To search for documents from the “Students” collection based on certain search criteria.

**Input:** Check the documents in the “Students” collection before proceeding.

```

C:\Windows\system32\cmd.exe - mongo
> db.Students.find({}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}

```

**Act:** Find the document wherein the “StudName” has value “Aryan David”.

```
db.Students.find({StudName:"Aryan David"});
```

**Outcome:**

```

C:\Windows\system32\cmd.exe - mongo
> db.Students.find({StudName:"Aryan David"});
{ "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }

```

To format the above output, use the pretty() method:

```
db.Students.find({StudName:"Aryan David"}).pretty();
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:"Aryan David"}).pretty();
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
```

*RDBMS equivalent:*

```
Select *
  From Students
  Where StudName like 'Aryan David';
```

```
SQL> select * from Students where StudName like 'Aryan David';
STUDR STUDNAME          GRADE HOBBIES
-----  -----
3      Aryan David        VII   Chess
SQL>
```

---

**Objective:** To display only the StudName from all the documents of the Student's collection. The identifier “\_id” should be suppressed and NOT displayed.

**Act:**

```
db.Students.find({}, {StudName:1, _id:0});\
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({}, {StudName:1, _id:0});
{ "StudName" : "Michelle Jacintha" }
{ "StudName" : "Aryan David" }
{ "StudName" : "Hersch Gibbs" }
{ "StudName" : "Vamsi Bapat" }
{ "StudName" : "Mabel Mathews" }
```

*RDBMS equivalent:*

```
Select StudName
  From Students;
```

```
SQL> select StudName from Students;
STUDNAME
-----
Michelle Jacintha
Aryan David
Mabel Mathews
Hersch Gibbs
Vamsi Bapat
SQL>
```

---

**Objective:** To display only the StudName and Grade from all the documents of the Students collection. The identifier `_id` should be suppressed and NOT displayed.

*Act:*

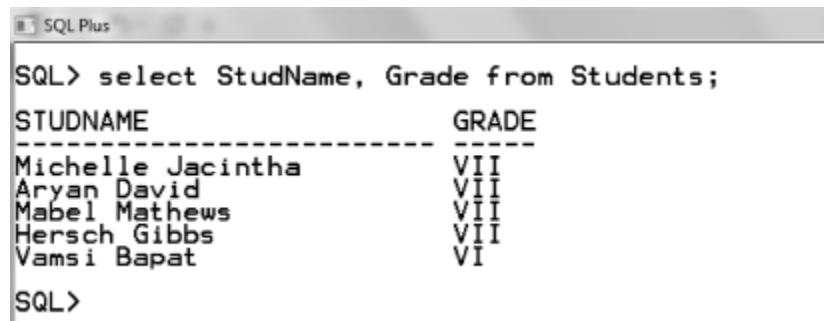
```
db.Students.find({}, {StudName:1, Grade:1, _id:0});
```

*Outcome:*

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({}, {StudName:1, Grade:1, _id:0});
{ "StudName" : "Michelle Jacintha", "Grade" : "VII" }
{ "Grade" : "VII", "StudName" : "Aryan David" }
{ "Grade" : "VII", "StudName" : "Hersch Gibbs" }
{ "StudName" : "Vamsi Bapat", "Grade" : "VI" }
{ "StudName" : "Mabel Mathews", "Grade" : "VII" }
```

### *RDBMS equivalent:*

```
Select StudName, Grade  
      From Students;
```



SQL> select StudName, Grade from Students;

| STUDNAME          | GRADE |
|-------------------|-------|
| Michelle Jacintha | VII   |
| Aryan David       | VII   |
| Mabel Mathews     | VII   |
| Hersch Gibbs      | VII   |
| Vamsi Bapat       | VI    |

SQL>

**Objective:** To display the StudName, Grade as well the identifier \_id from the document of the Students collection where the \_id column is 1.

**Act:**

```
db.Students.find({_id:1},{StudName:1,Grade:1});
```

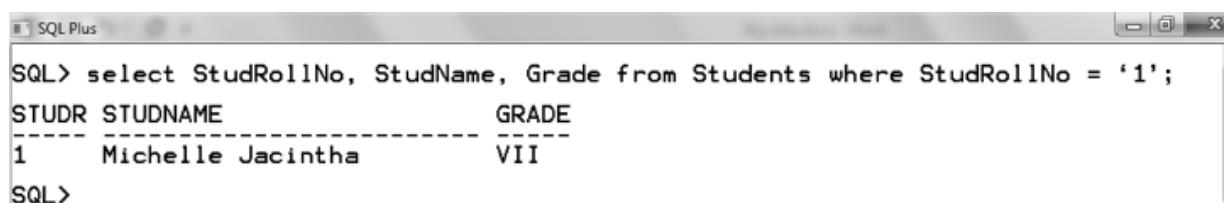
### *Outcome:*



```
C:\windows\system32\cmd.exe - mongo  
> db.Students.find({_id:1},{StudName:1,Grade:1});  
> { "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII" }
```

### *RDBMS equivalent:*

```
Select StudRollNo, StudName, Grade  
      From Students  
      Where StudRollNo = '1';
```



SQL> select StudRollNo, StudName, Grade from Students where StudRollNo = '1';

| STUDR | STUDNAME          | GRADE |
|-------|-------------------|-------|
| 1     | Michelle Jacintha | VII   |

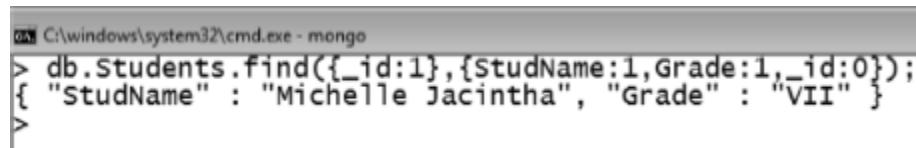
SQL>

**Objective:** To display the StudName and Grade from the document of the Students collection where the \_id column is 1. The \_id field should NOT be displayed.

**Act:**

```
db.Students.find({_id:1},{StudName:1,Grade:1,_id:0});
```

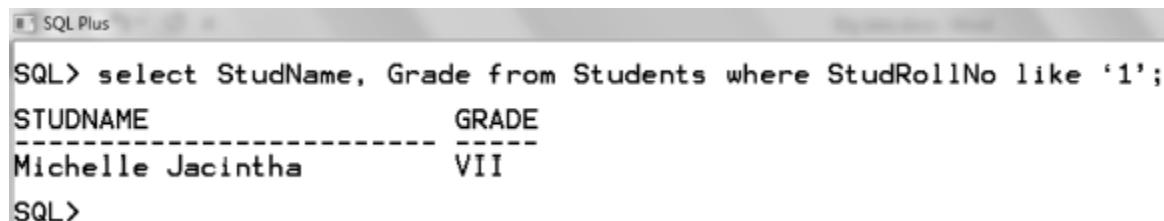
**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:1},{StudName:1,Grade:1,_id:0});
{ "StudName" : "Michelle Jacintha", "Grade" : "VII" }
```

**RDBMS equivalent:**

```
Select StudName, Grade
  From Students
    Where StudRollNo like '1';
```



```
SQL> select StudName, Grade from Students where StudRollNo like '1';
STUDNAME          GRADE
-----  -----
Michelle Jacintha    VII
SQL>
```

---

**Relational operators available to use in the search criteria**

|       |                            |
|-------|----------------------------|
| \$eq  | → equal to                 |
| \$ne  | → not equal to             |
| \$gte | → greater than or equal to |
| \$lte | → less than or equal to    |
| \$gt  | → greater than             |
| \$lt  | → less than                |

---

**Objective:** To find those documents where the Grade is set to 'VII'.

**Act:**

```
db.Students.find({Grade:{$eq:'VII'}}).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Grade:{$eq:'VII'}}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
```

**RDBMS equivalent:**

```
Select *
  From Students
  Where Grade like 'VII';
```

```
SQL> select * from Students where Grade like 'VII';
STUDR STUDNAME          GRADE HOBBIES
----- -----
1      Michelle Jacintha    VII  Internet surfing
3      Aryan David          VII  Chess
2      Mabel Mathews        VII  Baseball
4      Hersch Gibbs         VII  Graffiti
SQL>
```

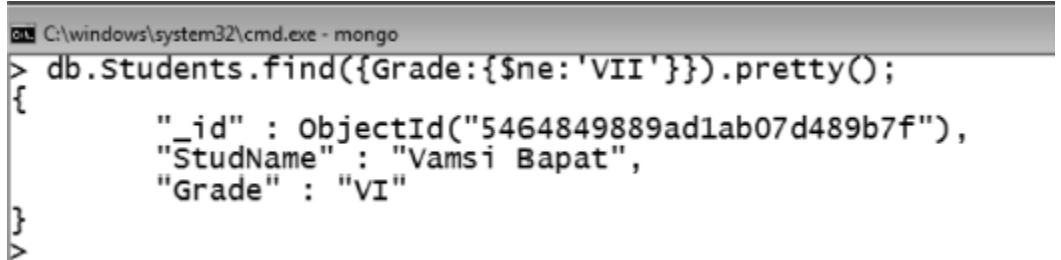
---

**Objective:** To find those documents where the Grade is NOT set to 'VII'

**Act:**

```
db.Students.find({Grade:{$ne:'VII'}}).pretty();
```

**Outcome:**

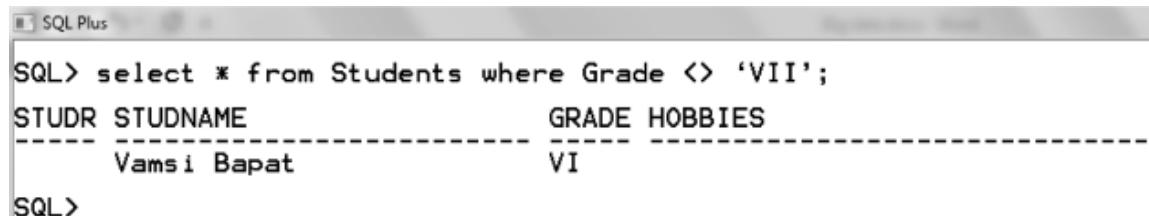


```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Grade:{$ne:'VII'}}).pretty();
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
>
```

There is just one document that meets the above criteria of Grade NOT EQUAL to 'VII'.

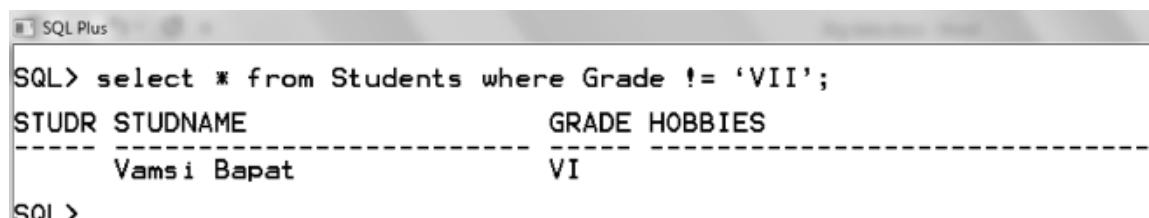
**RDBMS equivalent:**

```
Select *
  From Students
    Where Grade <> 'VII';
```



```
SQL> select * from Students where Grade <> 'VII';
STUDR STUDNAME          GRADE HOBBIES
----- -----
Vamsi Bapat              VI
SQL>
```

OR



```
SQL> select * from Students where Grade != 'VII';
STUDR STUDNAME          GRADE HOBBIES
----- -----
Vamsi Bapat              VI
SQL>
```

---

**Objective:** To find those documents from the Students collection where the Hobbies is set to either 'Chess' or 'Skating'.

**Act:**

```
db.Students.find ({Hobbies :{ $in: ['Chess', 'Skating']}}).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find ({Hobbies :{ $in: ['Chess','Skating']}}).pretty ();
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
```

**RDBMS equivalent:**

```
Select *
  From Students
    Where Hobbies in ('Chess', 'Skating');
```

```
SQL> select * from Students where Hobbies in ('Chess', 'Skating');
STUDR STUDNAME          GRADE HOBBIES
-----  -----
3      Aryan David        VII   Chess
SQL>
```

---

**Objective:** To find those documents from the Students collection where the Hobbies is set neither to 'Chess' nor to 'Skating'.

**Act:**

```
db.Students.find ({Hobbies :{ $nin: ['Chess','Skating']}}).pretty ();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find ({Hobbies :{ $nin: ['Chess','Skating']}}).pretty ();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}

{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}

{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}

{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
>
```

*RDBMS equivalent:*

```
Select *
  From Students
    Where Hobbies not in ('Chess', 'Skating');
```

```
SQL> select * from Students where Hobbies not in ('Chess', 'Skating');
STUDR STUDNAME          GRADE HOBBIES
----- ----- -----
1      Michelle Jacintha    VII   Internet surfing
2      Mabel Mathews       VII   Baseball
4      Hersch Gibbs        VII   Graffiti
SQL>
```

---

**Objective:** To find those documents from the Students collection where the Hobbies is set to 'Graffiti' and the StudName is set to 'Hersch Gibbs' (AND condition).

*Act:*

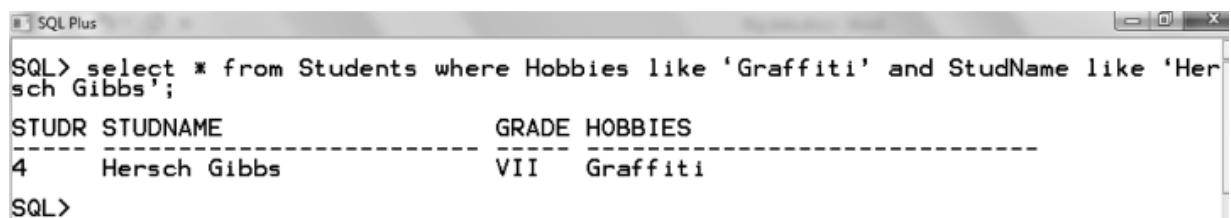
```
db.Students.find({Hobbies:'Graffiti', StudName: 'Hersch Gibbs'}).pretty();
```

*Outcome:*

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Hobbies:'Graffiti', StudName: 'Hersch Gibbs'}).pretty();
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
>
```

*RDBMS equivalent:*

```
Select *
  From Students
    Where Hobbies like 'Graffiti' and StudName like 'Hersch Gibbs';
```



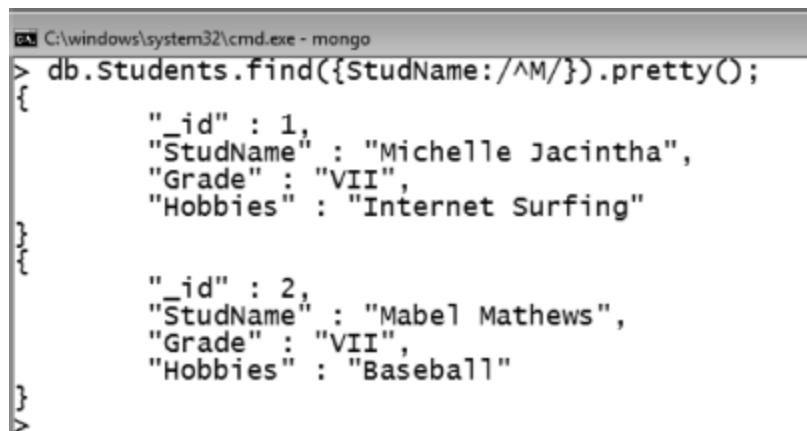
| STUDR | STUDNAME     | GRADE | HOBBIES  |
|-------|--------------|-------|----------|
| 4     | Hersch Gibbs | VII   | Graffiti |

**Objective:** To find documents from the Students collection where the StudName begins with “M”.

**Act:**

```
db.Students.find({StudName:/^M/}).pretty();
```

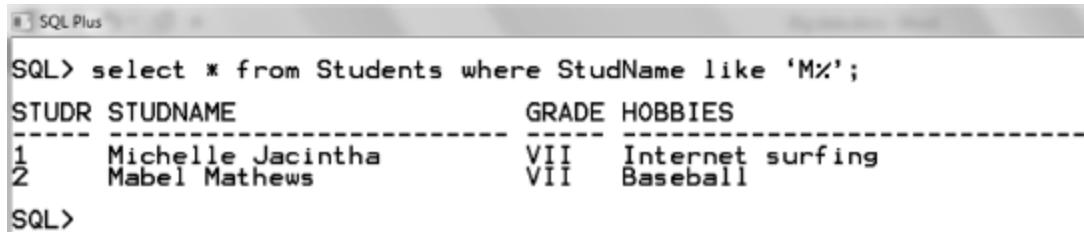
*Outcome:*



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:/^M/}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
```

*RDBMS equivalent:*

```
Select *
  From Students
    Where StudName like 'M%';
```



```
SQL> select * from Students where StudName like 'M%';
STUDR STUDNAME          GRADE HOBBIES
-----
1    Michelle Jacintha    VII   Internet surfing
2    Mabel Mathews       VII   Baseball
SQL>
```

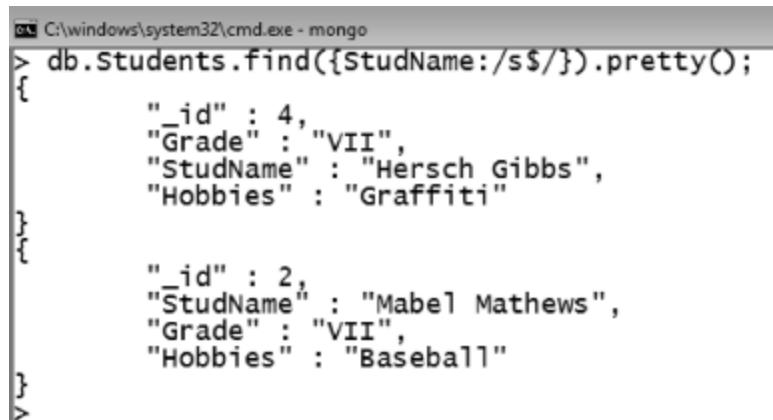
---

**Objective:** To find documents from the Students collection where the StudName ends in “s”.

**Act:**

```
db.Students.find({StudName:/s$/}).pretty();
```

**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:/s$/}).pretty();
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}>
```

**RDBMS equivalent:**

```
Select *
  From Students
    Where StudName like '%s';
```

```
SQL> select * from Students where StudName like '%s';
STUDR STUDNAME          GRADE HOBBIES
-----  -----
2      Mabel Mathews      VII   Baseball
4      Hersch Gibbs       VII   Graffiti
SQL>
```

**Objective:** To find documents from the Students collection where the StudName has an “e” in any position.

**Act:**

```
db.Students.find({StudName:/e/}).pretty();
```

**Or**

```
db.Students.find({StudName:/.*e.*/}).pretty();
```

**Or**

```
db.Students.find({StudName:{$regex:"e"}}).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:/e/}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
```

**RDBMS equivalent:**

```
Select *
  From Students
    Where StudName like '%e%';
```

SQL> select \* from Students where StudName like '%e%';

| STUDR | STUDNAME          | GRADE | HOBBIES          |
|-------|-------------------|-------|------------------|
| 1     | Michelle Jacintha | VII   | Internet surfing |
| 2     | Mabel Mathews     | VII   | Baseball         |
| 4     | Hersch Gibbs      | VII   | Graffiti         |

SQL>

**Objective:** To find documents from the Students collection where the StudName ends in “a”.

**Act:**

```
db.Students.find({StudName:{$regex:"a$"}}).pretty();
```

**Outcome:**

```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find({StudName:{$regex:"a$"}}).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}>
```

**RDBMS equivalent:**

```
Select *
  From Students
    Where StudName like '%a';
```

SQL> select \* from Students where StudName like '%a';

| STUDR | STUDNAME          | GRADE | HOBBIES          |
|-------|-------------------|-------|------------------|
| 1     | Michelle Jacintha | VII   | Internet surfing |

SQL>

## Dealing with NULL Values

**Objective:** To add a new field with null value in existing documents (`_id:3` and `_id:4`) of Students collection.

**Input:** Before we execute the commands to update documents with a null value in a column, let us first view the two documents.

```
db.Students.find({$or:[{_id:3},{_id:4}]})
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({$or:[{_id:3},{_id:4}]});
{
  "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }
{
  "_id" : 4, "Grade" : "VII", "StudName" : "Hersch Gibbs", "Hobbies" : "Graffiti" }
```

**Act:** Update the documents with NULL values in the “Location” column.

```
db.Students.update({_id:3},{$set:{Location:null}});
db.Students.update({_id:4},{$set:{Location:null}});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:3},{$set:{Location:null}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.Students.update({_id:4},{$set:{Location:null}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**RDBMS equivalent:**

```
Update Students
  Set Location = null
    Where StudRollNo in ('3', '4');
```

```
SQL> update Students set Location = null where StudRollNo in ('3','4');
2 rows updated.
SQL>
```

**Outcome:** To search for Null values in Location column.

```
db.Students.find({Location:{'$eq:null'}});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Location:{$eq:null}});
{
  "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII", "Hobbies" : "Internet Surfing" },
  {"_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess", "Location" : null },
  {"_id" : 4, "Grade" : "VII", "StudName" : "Hersch Gibbs", "Hobbies" : "Graffiti", "Location" : null },
  {"_id" : ObjectId("5464849889adlab07d489b7f"), "StudName" : "Vamsi Bapat", "Grade" : "VI" },
  {"_id" : 2, "StudName" : "Mabel Mathews", "Grade" : "VII", "Hobbies" : "Baseball" }
>
```

The above statement displays documents which either have null values in the location column or do not have the location column at all.

### ***RDBMS equivalent:***

```
Select *
  From Students
  Where Location is Null;
```

```
SQL> select * from Students where Location is NULL;
STUDR STUDNAME          GRADE HOBBIES          LOCATION
-----+-----+-----+-----+-----+-----+
1    Michelle Jacintha    VII   Internet surfing
3    Aryan David          VII   Chess
2    Mabel Mathews        VII   Baseball
4    Hersch Gibbs         VII   Graffiti
5    Vamsi Bapat          VI    null
SQL>
```

***Objective:*** To remove “Location” field having “NULL” values from the documents (\_id:3 and \_id:4) from the Students collection.

***Input:*** Document from the “Students” collection having “NULL” values in the “Location” column.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Location:{$eq:null}});
{
  "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII", "Hobbies" : "Internet Surfing" },
  {"_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess", "Location" : null },
  {"_id" : 4, "Grade" : "VII", "StudName" : "Hersch Gibbs", "Hobbies" : "Graffiti", "Location" : null },
  {"_id" : ObjectId("5464849889adlab07d489b7f"), "StudName" : "Vamsi Bapat", "Grade" : "VI" },
  {"_id" : 2, "StudName" : "Mabel Mathews", "Grade" : "VII", "Hobbies" : "Baseball" }
>
```

### ***Act:***

```
db.Students.update({_id:3}, {$unset:{Location:null}});
db.Students.update({_id:4}, {$unset:{Location:null}});
```

```
C:\windows\system32\cmd.exe - mongo
> db.Students.update({_id:3},{$unset:{Location:null}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
> db.Students.update({_id:4},{$unset:{Location:null}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 0 })
>
```

**Outcome:** Let us confirm if the changes have been made by running find method on the Students collection.

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({})
  "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII", "Hobbies" : "Internet Surfing" }
  "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }
  "_id" : 4, "Grade" : "VII", "StudName" : "Hersch Gibbs", "Hobbies" : "Graffiti" }
  "_id" : ObjectId("5464849889ad1ab07d489b7f"), "StudName" : "Vamsi Bapat", "Grade" : "VI" }
  "_id" : 2, "StudName" : "Mabel Mathews", "Grade" : "VII", "Hobbies" : "Baseball" }
```

## Count, Limit, Sort, and Skip

**Objective:** To find the number of documents in the Students collection.

**Act:**

```
db.Students.count()
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.count()
5
>
```

**Objective:** To find the number of documents in the Students collection wherein the Grade is VII.

**Act:**

```
db.Students.count({Grade:"VII"});
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.count({Grade:"VII"});
4
>
```

---

**Objective:** To retrieve the first 3 documents from the Students collection wherein the Grade is VII.

**Act:**

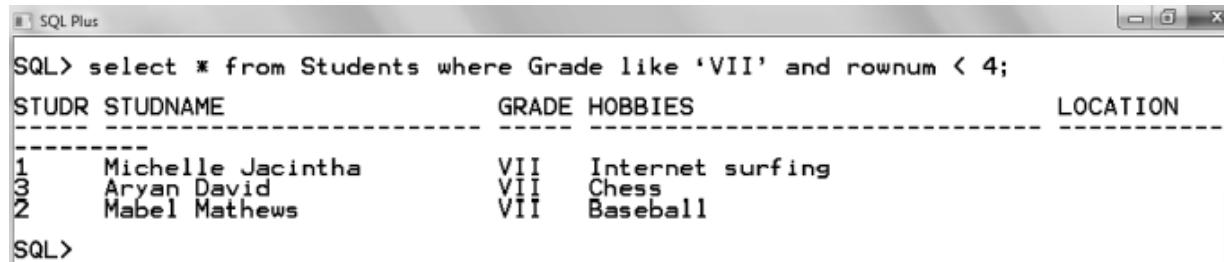
```
db.Students.find({Grade:"VII"}).limit(3).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({Grade:"VII"}).limit(3).pretty();
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
```

**RDBMS equivalent:**

```
Select *
  From Students
    Where Grade like 'VII' and rownum < 4;
```



SQL> select \* from Students where Grade like 'VII' and rownum < 4;

| STUDR | STUDNAME          | GRADE | HOBBIES          | LOCATION |
|-------|-------------------|-------|------------------|----------|
| 1     | Michelle Jacintha | VII   | Internet surfing |          |
| 3     | Aryan David       | VII   | Chess            |          |
| 2     | Mabel Mathews     | VII   | Baseball         |          |

SQL>

**Objective:** To sort the documents from the Students collection in the ascending order of StudName.

**Act:**

```
db.Students.find().sort({StudName:1}).pretty();
```

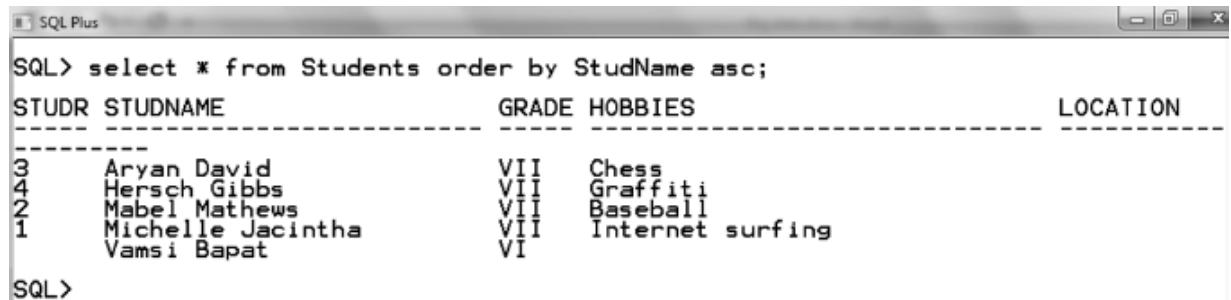
**Outcome:**



```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find().sort({StudName:1}).pretty();
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
>
```

**RDBMS equivalent:**

```
Select *
  From Students
    Order by StudName asc;
```



SQL> select \* from Students order by StudName asc;

| STUDR | STUDNAME          | GRADE | HOBBIES          | LOCATION |
|-------|-------------------|-------|------------------|----------|
| 3     | Aryan David       | VII   | Chess            |          |
| 4     | Hersch Gibbs      | VII   | Graffiti         |          |
| 2     | Mabel Mathews     | VII   | Baseball         |          |
| 1     | Michelle Jacintha | VII   | Internet surfing |          |
|       | Vamsi Bapat       | VI    |                  |          |

SQL>

---

**Objective:** To sort the documents from the Students collection in the descending order of StudName.

**Act:**

```
db.Students.find().sort({StudName:-1}).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().sort({StudName:-1}).pretty();
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
>
```

*RDBMS equivalent:*

```
Select *
  From Students
    Order by StudName desc;
```

| STUDR | STUDNAME          | GRADE | HOBBIES          | LOCATION |
|-------|-------------------|-------|------------------|----------|
| 1     | Vamsi Bapat       | VI    |                  |          |
| 2     | Michelle Jacintha | VII   | Internet surfing |          |
| 4     | Mabel Mathews     | VII   | Baseball         |          |
| 3     | Hersch Gibbs      | VII   | Graffiti         |          |
|       | Aryan David       | VII   | Chess            |          |

**Objective:** To sort the documents from the Students collection first on Grade in ascending order and then on Hobbies in descending order.

**Act:**

```
db.Students.find().sort({Grade:1, Hobbies:-1}).pretty();
```

### Outcome:

```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find().sort({Grade:1, Hobbies:-1}).pretty();
{
  "_id" : ObjectId("5464849889adlab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}

{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}

{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}

{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}

{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
```

### RDBMS equivalent:

```
Select *
  From Students
    Order by Grade asc, hobbies desc;
```

```
SQL> select * from Students order by Grade asc, hobbies desc;
STUDR STUDNAME          GRADE HOBBIES          LOCATION
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1     Vamsi Bapat          VI
2     Michelle Jacintha    VII    Internet surfing
4     Hersch Gibbs         VII    Graffiti
3     Aryan David          VII    Chess
2     Mabel Mathews        VII    Baseball
SQL>
```

**Objective:** To sort the documents from the Students collection first on Grade in ascending order and then on Hobbies in ascending order.

**Act:**

```
db.Students.find().sort({Grade:1, Hobbies:1}).pretty();
```

**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().sort({Grade:1, Hobbies:1}).pretty();
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 3,
  "Grade" : "VII",
  "StudName" : "Aryan David",
  "Hobbies" : "Chess"
}
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
>
```

**RDBMS equivalent:**

```
Select *
  From Students
    Order by Grade asc, Hobbies asc;
```

```
SQL> select * from Students order by Grade asc, Hobbies asc;
STUDR STUDNAME          GRADE HOBBIES          LOCATION
-----  -----
2      Vamsi Bapat        VI
3      Mabel Mathews      VII  Baseball
3      Aryan David        VII  Chess
4      Hersch Gibbs       VII  Graffiti
1      Michelle Jacintha  VII  Internet surfing

SQL>
```

**Objective:** To skip the first 2 documents from the Students collection.

**Act:**

```
db.Students.find().skip(2).pretty();
```

**Outcome:**

```
C:\Windows\system32\cmd.exe - mongo
> db.Students.find().skip(2).pretty();
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}>
```

**RDBMS equivalent:**

```
Select StudRollNo, StudName, Grade, Hobbies
  From (Select StudRollNo, StudName, Grade, Hobbies, RowNum as TheRow-
        Num From Students)
        Where TheRowNum > 2;
```

```
SQL> Select StudRollNo, StudName, Grade, Hobbies from (Select StudRollNo, StudName, Grade, Hobbies, RowNum as therownum from Students) where theRownum > 2;
STUDR STUDNAME          GRADE HOBBIES
----- -----
2      Mabel Mathews      VII   Baseball
4      Hersch Gibbs       VII   Graffiti
5      Vamsi Bapat        VI

SQL>
```

**Objective:** To sort the documents from the Students collection and skip the first document from the output.

**Act:**

```
db.Students.find().skip(1).pretty().sort({StudName:1});
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().skip(1).pretty().sort({StudName:1});
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
{
  "_id" : 1,
  "StudName" : "Michelle Jacintha",
  "Grade" : "VII",
  "Hobbies" : "Internet Surfing"
}
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
>
```

**RDBMS equivalent:**

```
Select StudRollNo, StudName, Grade, Hobbies
  From (Select StudRollNo, StudName, Grade, Hobbies, RowNum as The-
RowNum From Students)
        Where TheRowNum > 1
        Order by StudName;
```

```
SQL> Select StudRollNo, StudName, Grade, Hobbies from (Select StudRollNo, StudName, Grade, Hobbies, Rownum as therownum from Students) where therownum >1 order by StudName;
STUDR STUDNAME          GRADE HOBBIES
----- -----
3      Aryan David      VII   Chess
4      Hersch Gibbs     VII   Graffiti
2      Mabel Mathews    VII   Baseball
Vamsi Bapat          VI

SQL>
```

**Objective:** To display the last 2 records from the Students collection

**Act:**

```
db.Students.find().pretty().skip(db.Students.count() - 2);
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty().skip(db.Students.count() - 2);
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}

{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
```

**Objective:** To retrieve the third, fourth and fifth document from the Students collection

**Act:**

```
db.Students.find().pretty().skip(2).limit(3);
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find().pretty().skip(2).limit(3);
{
  "_id" : 4,
  "Grade" : "VII",
  "StudName" : "Hersch Gibbs",
  "Hobbies" : "Graffiti"
}
{
  "_id" : ObjectId("5464849889ad1ab07d489b7f"),
  "StudName" : "Vamsi Bapat",
  "Grade" : "VI"
}
{
  "_id" : 2,
  "StudName" : "Mabel Mathews",
  "Grade" : "VII",
  "Hobbies" : "Baseball"
}
>
```

#### 4.4.4 Arrays

**Objective:** To create a collection by the name “food” and then insert documents into the “food” collection. Each document should have a “fruits” array.

**Act:**

```
db.food.insert({_id:1,fruits:[ 'banana', 'apple', 'cherry' ] })
db.food.insert({_id:2,fruits:[ 'orange', 'butterfruit', 'mango' ]})
db.food.insert({_id:3,fruits:[ 'pineapple', 'strawberry', 'grapes' ]});
db.food.insert({_id:4,fruits:[ 'banana', 'strawberry', 'grapes' ]});
db.food.insert({_id:5,fruits:[ 'orange', 'grapes' ]});
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.insert({_id:1,fruits:[ 'banana', 'apple', 'cherry' ] })
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:2,fruits:[ 'orange', 'butterfruit', 'mango' ]})
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:3,fruits:[ 'pineapple', 'strawberry', 'grapes' ]});
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:4,fruits:[ 'banana', 'strawberry', 'grapes' ]});
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:5,fruits:[ 'orange', 'grapes' ]});
WriteResult({ "nInserted" : 1 })
>
```

**Outcome:** Let us check if these documents are now in the “food” collection.

```
db.food.find({})
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({})
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
```

---

**Objective:** To find those documents from the “food” collection which has the “fruits array” constituted of “banana”, “apple” and “cherry”.

**Act:**

```
db.food.find({fruits:[ 'banana', 'apple', 'cherry' ]}).pretty()
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({fruits:[ 'banana', 'apple', 'cherry' ]}).pretty()
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
```

---

**Objective:** To find those documents from the “food” collection which have the “fruits” array having “banana” as an element.

**Act:**

```
db.food.find({fruits: 'banana'})
```

**Outcome:**

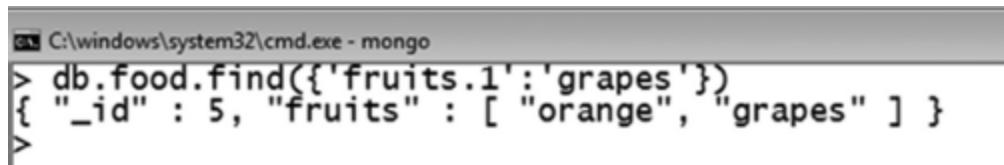
```
C:\windows\system32\cmd.exe - mongo
> db.food.find({fruits: 'banana'})
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
```

**Objective:** To find those documents from the “food” collection which have the “fruits” array having “grapes” in the first index position. The index position begins at 0.

**Act:**

```
db.food.find({ 'fruits.1' : 'grapes' })
```

**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({ 'fruits.1' : 'grapes' })
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
```

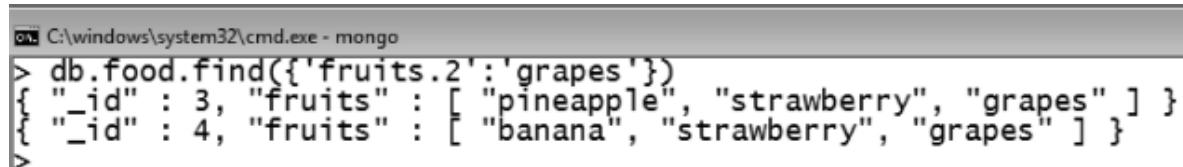
---

**Objective:** To find those documents from the “food” collection where “grapes” is present in the 2nd index position of the “fruits” array.

**Act:**

```
db.food.find({ 'fruits.2' : 'grapes' })
```

**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({ 'fruits.2' : 'grapes' })
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
```

---

**Objective:** To find those documents from the “food” collection where the size of the array is two. The size implies that the array holds only 2 values.

**Act:**

```
db.food.find({ "fruits" : { $size: 2 } })
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({"fruits":{$size:2}})
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

---

**Objective:** To find those documents from the “food” collection where the size of the array is three. The size implies that the array holds only 3 values

**Act:**

```
db.food.find({"fruits":{$size:3}})
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({"fruits":{$size:3}});
[{"_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] },
 {"_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] },
 {"_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] },
 {"_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }]
>
```

---

**Objective:** To find the document with (\_id: 1) from the “food” collection and display the first two elements from the array “fruits”.

**Act:**

```
db.food.find({_id:1}, {"fruits":{$slice:2}})
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:1}, {"fruits":{$slice:2}})
{ "_id" : 1, "fruits" : [ "banana", "apple" ] }
>
```

---

**Objective:** To find all documents from the “food” collection which have elements “orange” and “grapes” in the array “fruits”.

**Act:**

```
db.food.find ({"fruits": {"$all: ["orange", "grapes"]}}).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find ({"fruits": {"$all: ["orange", "grapes"]}}).pretty();
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
```

---

**Objective:** To find those documents from the “food” collection which have the element “orange” in the 0<sup>th</sup> index position in the array, “fruits”.

**Act:**

```
db.food.find ({"fruits.0": "orange"}).pretty();
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find ({"fruits.0": "orange"}).pretty();
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
```

---

**Objective:** To find the document with (\_id: 1) from the “food” collection and display two elements from the array “fruits”, starting with the element at 0<sup>th</sup> index position.

**Act:**

```
db.food.find({_id:1}, {"fruits":{$slice:[0,2]}})
```

**Outcome:**

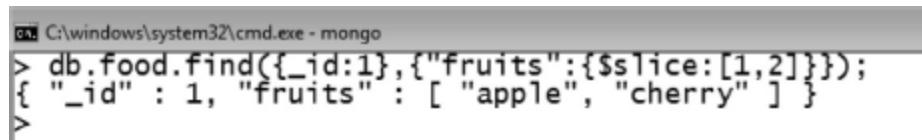
```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:1}, {"fruits":{$slice:[0,2]}})
{ "_id" : 1, "fruits" : [ "banana", "apple" ] }
```

**Objective:** To find the document with (\_id: 1) from the “food” collection and display two elements from the array “fruits”, starting with the element at 1st index position.

**Act:**

```
db.food.find({_id:1}, {"fruits":{$slice:[1,2]}})
```

**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:1}, {"fruits":{$slice:[1,2]}});
{ "_id" : 1, "fruits" : [ "apple", "cherry" ] }
```

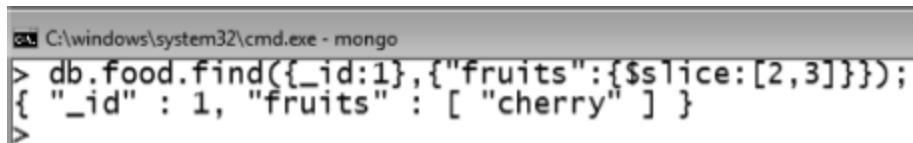
---

**Objective:** To find the document with (\_id: 1) from the “food” collection and display three elements from the array “fruits”, starting with the element at 2nd index position. Since we have only 3 elements in the array “fruits” for the document with \_id:1, it displays only one element, the element at 2nd index position, that is, “cherry”

**Act:**

```
db.food.find({_id:1}, {"fruits":{$slice:[2,3]}})
```

**Outcome:**



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:1}, {"fruits":{$slice:[2,3]}});
{ "_id" : 1, "fruits" : [ "cherry" ] }
```

---

## Update on the Array

Before we begin the update operations on the “fruits” array of the documents of “food” collection, let us take a look at the documents that we have in the “food” collection:

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({})
[{"_id": 1, "fruits": ["banana", "apple", "cherry"]},
 {"_id": 2, "fruits": ["orange", "butterfruit", "mango"]},
 {"_id": 3, "fruits": ["pineapple", "strawberry", "grapes"]},
 {"_id": 4, "fruits": ["banana", "strawberry", "grapes"]},
 {"_id": 5, "fruits": ["orange", "grapes"]}]
```

**Objective:** To update the document with “\_id:4” and replace the element present in the 1st index position of the “fruits” array with “apple”.

**Act:**

```
db.food.update({_id:4}, {$set:{'fruits.1': 'apple'}})
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4}, {$set:{'fruits.1': 'apple'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

**Outcome:** Let us take a look at how this update has changed our document.

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4})
[{"_id": 4, "fruits": ["banana", "apple", "grapes"]}]
```

---

**Objective:** To update the document with “\_id:1” and replace the element “apple” of the “fruits” array with “An apple”.

**Act:**

```
db.food.update({_id:1, 'fruits':'apple'}, {$set:{'fruits.$': 'An apple'}})
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:1, 'fruits':'apple'}, {$set:{'fruits.$': 'An apple'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

**Outcome:** The document after update looks as follows.

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:1});
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
>
```

---

**Objective:** To update the document with “\_id:2” and push new key-value pairs in the “fruits” array.

**Act:**

```
db.food.update({_id:2}, {$push:{price:{orange:60,butterfruit:200,
mango:120}}})
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:2},{$push:{price:{orange:60,butterfruit:200,mango:120}}});
writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:**

```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{
  "_id" : 1,
  "fruits" : [
    "banana",
    "An apple",
    "cherry"
  ]
}
{
  "_id" : 3,
  "fruits" : [
    "pineapple",
    "strawberry",
    "grapes"
  ]
}
{
  "_id" : 4,
  "fruits" : [
    "banana",
    "apple",
    "grapes"
  ]
}
{
  "_id" : 5,
  "fruits" : [
    "orange",
    "grapes"
  ]
}
{
  "_id" : 2,
  "fruits" : [
    "orange",
    "butterfruit",
    "mango"
  ],
  "price" : [
    {
      "orange" : 60,
      "butterfruit" : 200,
      "mango" : 120
    }
  ]
}
```

---

### **Further Updates to the Array “fruits”**

Before we do the updates to the documents in the food collection, let us look at the current state:

```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{
  "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{
  "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{
  "_id" : 4, "fruits" : [ "banana", "apple", "grapes" ] }
{
  "_id" : 5, "fruits" : [ "orange", "grapes" ] }

  "_id" : 2,
  "fruits" : [
    "orange",
    "butterfruit",
    "mango"
  ],
  "price" : [
    {
      "orange" : 60,
      "butterfruit" : 200,
      "mango" : 120
    }
  ]
}
```

---

**Objective:** To update the document with “\_id:4” by adding an element “orange” to the list of elements in the array, “fruits”.

**Act:**

```
db.food.update({_id:4}, {$addToSet:{fruits:"orange"} });
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4}, {$addToSet:{fruits:"orange"} });
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** The result after the execution of the statement is as follows.

```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
[{"_id": 1, "fruits": ["banana", "An apple", "cherry"]},
 {"_id": 3, "fruits": ["pineapple", "strawberry", "grapes"]},
 {"_id": 4, "fruits": ["banana", "apple", "grapes", "orange"]},
 {"_id": 5, "fruits": ["orange", "grapes"]},
 {"_id": 2, "fruits": ["orange", "butterfruit", "mango"]},
 {"price": [
   {
     "orange": 60,
     "butterfruit": 200,
     "mango": 120
   }
]}
>
```

---

**Objective:** To update the document with “\_id:4” by popping an element from the list of elements present in the array “fruits”. The element popped is the one from the end of the array.

**Act:**

```
db.food.update({_id:4}, {$pop:{fruits:1}});
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4}, {$pop:{fruits:1}});
writeResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
>
```

**Outcome:** The “food” collection after the execution of the statement is as follows.

```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
[{"_id": 1, "fruits": ["banana", "An apple", "cherry"]},
 {"_id": 3, "fruits": ["pineapple", "strawberry", "grapes"]},
 {"_id": 4, "fruits": ["banana", "apple", "grapes"]},
 {"_id": 5, "fruits": ["orange", "grapes"]},
 {"_id": 2, "fruits": ["orange", "butterfruit", "mango"]},
 {"price": [
   {"orange": 60, "butterfruit": 200, "mango": 120}
 ]}
```

**Objective:** To update the document with “`_id:4`” by popping an element from the list of elements present in the array “`fruits`”. The element popped is the one from the beginning of the array

**Act:**

```
db.food.update({_id:4}, {$pop:{fruits:-1}});
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4}, {$pop:{fruits:-1}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** The “`food`” collection after the execution of the above update statement is

```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{
  "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{
  "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{
  "_id" : 4, "fruits" : [ "apple", "grapes" ] }
{
  "_id" : 5, "fruits" : [ "orange", "grapes" ] }

  "_id" : 2,
  "fruits" : [
    "orange",
    "butterfruit",
    "mango"
  ],
  "price" : [
    {
      "orange" : 60,
      "butterfruit" : 200,
      "mango" : 120
    }
  ]
}
>
```

**Objective:** To update the document with “\_id:3” by popping two elements from the list of elements present in the array “fruits”. The element popped are “pineapple” and “grapes”.

The document with “\_id:3”, before the update is

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:3});
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
```

**Act:**

```
db.food.update({_id:3}, {$pullAll:{fruits: [ 'pineapple', 'grapes' ]}});
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:3}, {$pullAll:{fruits: [ 'pineapple', 'grapes' ]}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** The document with “\_id:3” after the update is

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4});
{ "_id" : 4, "fruits" : [ "apple", "grapes" ] }
>
```

**Objective:** To update the documents having “banana” as an element in the array “fruits” and pop out the element “banana” from those documents.

The “food” collection before the update is

```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{
  "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ]
  "_id" : 3, "fruits" : [ "strawberry" ]
  "_id" : 4, "fruits" : [ "apple", "grapes" ]
  "_id" : 5, "fruits" : [ "orange", "grapes" ]
}

  "_id" : 2,
  "fruits" : [
    "orange",
    "butterfruit",
    "mango"
    ]

  "price" : [
    {
      "orange" : 60,
      "butterfruit" : 200,
      "mango" : 120
    }
  ]
}
>
```

**Act:**

```
db.food.update({fruits:'banana'}, { $pull:{fruits:'banana'}})
```

```
C:\windows\system32\cmd.exe - mongo
> db.food.update({fruits:'banana'}, { $pull:{fruits:'banana'}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** The “food” collection after the update is

```
on C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{
  "_id" : 1, "fruits" : [ "An apple", "cherry" ] }
  "_id" : 3, "fruits" : [ "strawberry" ] }
  "_id" : 4, "fruits" : [ "apple", "grapes" ] }
  "_id" : 5, "fruits" : [ "orange", "grapes" ] }

  "_id" : 2,
  "fruits" : [
    "orange",
    "butterfruit",
    "mango"
  ],
  "price" : [
    {
      "orange" : 60,
      "butterfruit" : 200,
      "mango" : 120
    }
  ]
}
```

**Objective:** To pull out an array element based on index position.

There is no direct way of pulling the array elements by looking up their index numbers. However, a workaround is available. The document with “\_id:4” in the food collection prior to the update is

```
on C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4}).pretty();
{
  "_id" : 4, "fruits" : [ "apple", "grapes" ] }
>
```

**Act:** The update statements are

```
db.food.update({_id:4}, {$unset : {"fruits.1" : null }});
db.food.update({_id:4}, {$pull : {"fruits" : null}});
```

```
on C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4}, {$unset : {"fruits.1" : null }});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.food.update({_id:4}, {$pull : {"fruits" : null}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

**Outcome:** After update, the document with \_id:4 in the food collection is

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4}).pretty();
{ "_id" : 4, "fruits" : [ "apple" ] }
>
```

---

## Operators at a Glance

Below we summarize all operators used above.

### Relational Operators to use in Queries

| Operator | Description                                                    |
|----------|----------------------------------------------------------------|
| \$eq     | Matches values equal to a specified value                      |
| \$lt     | Matches values less than a specified value                     |
| \$lte    | Matches values less than or equal to a specified value         |
| \$gt     | Matches values greater than a specified value                  |
| \$gte    | Matches values greater than or equal to a specified value      |
| \$in     | Matches any of the value specified                             |
| \$nin    | Does not match any of the value specified                      |
| \$ne     | Matches all the values that are not equal to a specified value |

### Logical Operators to use in Queries

| Operator | Description                                                               |
|----------|---------------------------------------------------------------------------|
| \$and    | Returns documents that match the criteria given in both the query clauses |
| \$or     | Returns documents that match criteria of either query clause              |
| \$not    | Returns documents that do not match the query [Inversion]                 |

| Operator | Description                                                |
|----------|------------------------------------------------------------|
| \$nor    | Returns documents that do not match both the query clauses |

#### Element Query Operators

| Operator | Description                                           |
|----------|-------------------------------------------------------|
| \$exists | Matches documents that contain the specified field    |
| \$type   | Matches documents based on the BSON type of the field |

#### Array Query Operators

| Operator    | Description                                                                    |
|-------------|--------------------------------------------------------------------------------|
| \$size      | Retrieves documents having the specified array size                            |
| \$all       | Matches documents containing all the elements specified in the query           |
| \$elemMatch | Retrieves documents if an array element satisfies all the conditions specified |

#### Field Update Operators

| Operator | Description                                                                |
|----------|----------------------------------------------------------------------------|
| \$set    | Sets the field value in the document                                       |
| \$unset  | Removes the specified field from the document                              |
| \$min    | Updates the value of the field only if it is less than the existing one    |
| \$max    | Updates the value of the field only if it is greater than the existing one |
| \$mul    | Multiplies the field value by the given value                              |

| Operator      | Description                                                                         |
|---------------|-------------------------------------------------------------------------------------|
| \$inc         | Increments the field value by the given value                                       |
| \$rename      | Renames a field                                                                     |
| \$setOnInsert | If a new document is inserted during an update, then it sets the value of the field |
| \$currentDate | Sets the field value to the current date                                            |

#### Array Update Operators

| Operator   | Description                                                                                     |
|------------|-------------------------------------------------------------------------------------------------|
| \$         | Updates the first element that matches the criteria                                             |
| \$push     | Adds elements to an array (duplicates can exist)                                                |
| \$addToSet | Adds elements to an array only if not present                                                   |
| \$pop      | Removes an element from an array (-1 to remove the first element, 1 to remove the last element) |

#### Array Update Modifiers

| Operator   | Description                                                                       |
|------------|-----------------------------------------------------------------------------------|
| \$slice    | Limits the array elements during the \$push operation                             |
| \$sort     | Orders array elements during the \$push operation                                 |
| \$each     | To add multiple elements to an array during either \$push or \$addToSet operation |
| \$position | Specifies the position in the array to insert elements during \$push operation    |

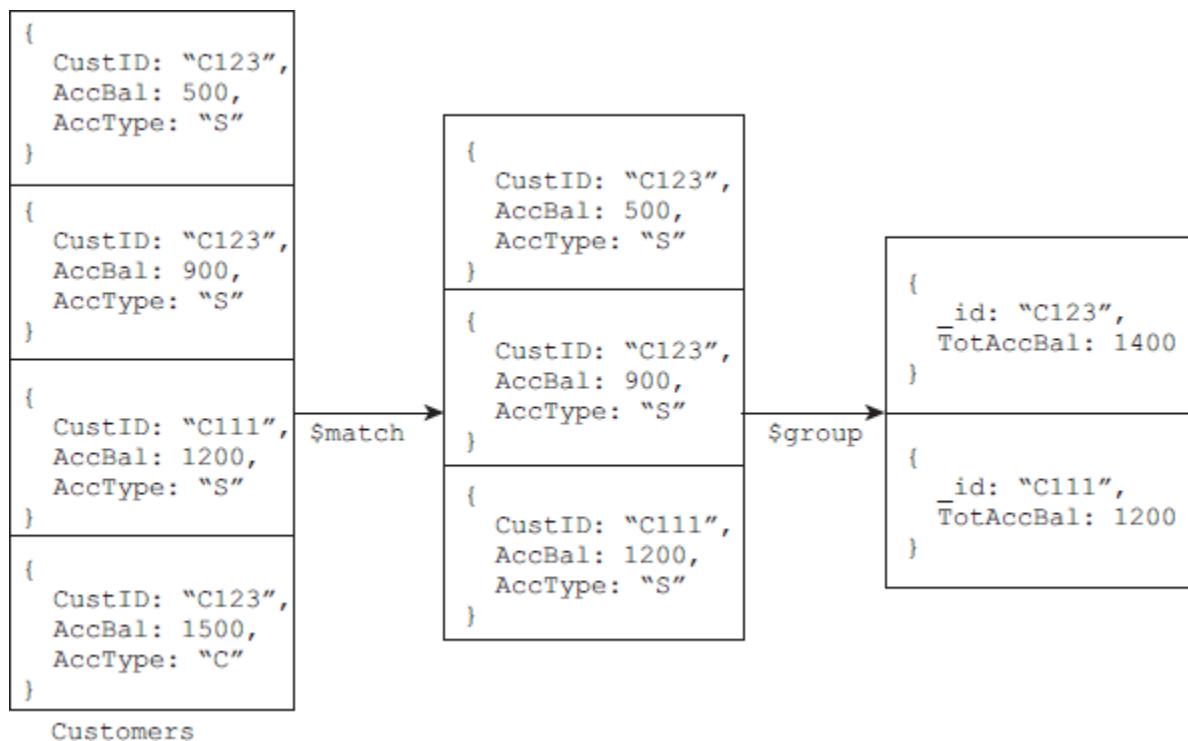
### 4.4.5 Aggregate Function

Let us look at how aggregation operations work in MongoDB.

1. The first step is to group values across multiple documents.

- The second step is to run a variety of aggregate operations (such as sum, average, minimum, maximum, count) on the grouped values to return a single result.

**Objective:** Consider the collection “Customers” as given below. It has four documents. We would like to filter out those documents where the “AccType” has a value other than “S”. After the filter, we should be left with three documents where the “AccType”: “S”. It is then required to group the documents on the basis of CustID and sum up the “AccBal” for each unique “CustID”. This is similar to the output received with group by clause in RDBMS. Once the groups have been formed (as per the example below, there will be only two groups: (a) “CustID” : “C123” and (b) “CustID” : “C111”. Filter and display that group where the “TotAccBal” column has a value greater than 1200.



**Figure 4.4** A customer collection with four documents.

Let us start off by creating the collection “Customers” with four documents as shown in [Figure 4.4](#).

```
db.Customers.insert([{"CustID": "C123", "AccBal": 500, "AccType": "S"},  
{"CustID": "C123", "AccBal": 900, "AccType": "S"},  
{"CustID": "C111", "AccBal": 1200, "AccType": "S"},  
{"CustID": "C123", "AccBal": 1500, "AccType": "C"}]);
```

```
C:\windows\system32\cmd.exe - mongo  
> db.Customers.insert([{"CustID": "C123", "AccBal": 500, "AccType": "S"}, {"CustID": "C123", "AccBal": 900, "AccType": "S"},  
... {"CustID": "C111", "AccBal": 1200, "AccType": "S"}, {"CustID": "C123", "AccBal": 1500, "AccType": "C"}]);  
BulkWriteResult[  
  {"writeErrors": [], "writeConcernErrors": [], "nInserted": 4, "nUpserted": 0, "nMatched": 0, "nModified": 0, "nRemoved": 0, "upserted": []}  
]  
>
```

*To confirm the presence of four documents in the “Customers” collection, use the following syntax:*

```
db.Customers.find().pretty();
```

```
C:\windows\system32\cmd.exe - mongo  
> db.Customers.find().pretty();  
{  
  "_id" : ObjectId("54993269f4263d0150bfa72c"),  
  "CustID" : "C123",  
  "AccBal" : 500,  
  "AccType" : "S"  
}  
{  
  "_id" : ObjectId("54993269f4263d0150bfa72d"),  
  "CustID" : "C123",  
  "AccBal" : 900,  
  "AccType" : "S"  
}  
{  
  "_id" : ObjectId("54993269f4263d0150bfa72e"),  
  "CustID" : "C111",  
  "AccBal" : 1200,  
  "AccType" : "S"  
}  
{  
  "_id" : ObjectId("54993269f4263d0150bfa72f"),  
  "CustID" : "C123",  
  "AccBal" : 1500,  
  "AccType" : "C"  
}  
>
```

*To group on “CustID” and compute the sum of “AccBal”, use the following syntax:*

```
db.Customers.aggregate( { $group : { _id : "$CustID", TotAccBal : { $sum :  
  "$AccBal" } } } );
```

```
C:\windows\system32\cmd.exe - mongo
> db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $sum : "$AccBal" } } });
> { "_id" : "C111", "TotAccBal" : 1200 }
> { "_id" : "C123", "TotAccBal" : 2900 }
```

*In order to first filter on “AccType:S” and then group it on “CustID” and then compute the sum of “AccBal”, use the following syntax:*

```
db.Customers.aggregate( { $match : {AccType : "S" } },
{ $group : { _id : "$CustID",TotAccBal : { $sum : "$AccBal" } } } );
```

```
C:\windows\system32\cmd.exe - mongo
> db.Customers.aggregate( { $match : {AccType : "S" } },
... { $group : { _id : "$CustID",TotAccBal : { $sum : "$AccBal" } } } );
{ "_id" : "C111", "TotAccBal" : 1200 }
{ "_id" : "C123", "TotAccBal" : 1400 }
>
```

*In order to first filter on “AccType:S”, then group it on “CustID”, then to compute the sum of “AccBal” and then filter those documents wherein the “TotAccBal” is greater than 1200, use the following syntax:*

```
db.Customers.aggregate( { $match : {AccType : "S" } },
{ $group : { _id : "$CustID",TotAccBal : { $sum : "$AccBal" } } },
{ $match : {TotAccBal : { $gt : 1200 } } } );
```

```
C:\windows\system32\cmd.exe - mongo
> db.Customers.aggregate( { $match : {AccType : "S" } },
... { $group : { _id : "$CustID",TotAccBal : { $sum : "$AccBal" } } },
... { $match : {TotAccBal : { $gt : 1200 } } } );
{ "_id" : "C123", "TotAccBal" : 1400 }
>
```

*To group on “CustID” and compute the average of the “AccBal” for each group.*

```
db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $avg : "$AccBal" } } } );
```

```
> db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $avg : "$AccBal" } } } );
{ "_id" : "C111", "TotAccBal" : 1200 }
{ "_id" : "C123", "TotAccBal" : 966.666666666666 }
```

*To group on “CustID” and determine the maximum “AccBal” for each group.*

```
db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $max : "$AccBal" } } } );
```

```
> db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $max : "$AccBal" } } } );  
> {  
  "_id" : "C111", "TotAccBal" : 1200  
}  
> {  
  "_id" : "C123", "TotAccBal" : 1500  
}
```

*To group on “CustID” and determine the maximum “AccBal” for each group.*

```
db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $min : "$AccBal" } } } );
```

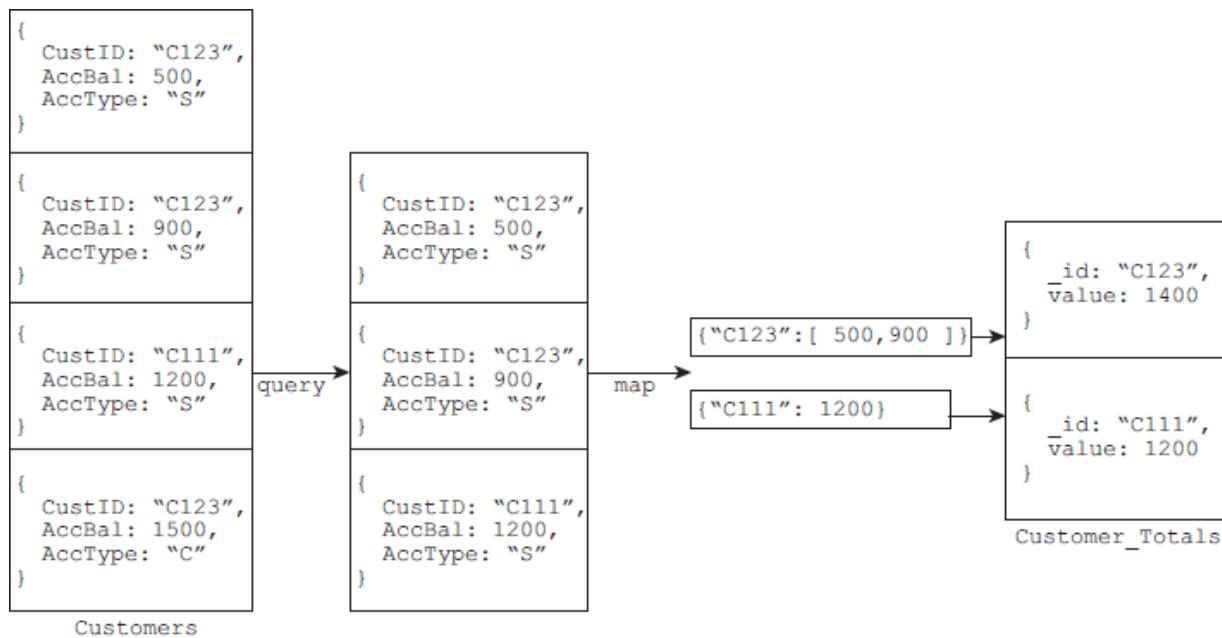
```
> db.Customers.aggregate( { $group : { _id : "$CustID",TotAccBal : { $min : "$AccBal" } } } );  
> {  
  "_id" : "C111", "TotAccBal" : 1200  
}  
> {  
  "_id" : "C123", "TotAccBal" : 500  
}
```

#### 4.4.6 MapReduce Framework

MongoDB also provides map-reduce operations to perform aggregation. In general, map-reduce operations have two phases:

1. a *map* stage that processes each document and *emits* one or more objects for each input document, and
2. *reduce* phase that combines the output of the map operation.

**Objective:** Consider the collection “Customers” below. There are four documents. Run a query to filter out those documents where the key “AccType” has a value other than “S”. Then for each unique CustID, prepare a list of AccBal values. For example, for CustID: “C123”, the AccBals is 500,900. This task will be assigned to the mapper function. The output from the mapper function serves as the input to the reducer function. The reducer function then aggregates the AccBal for each CustID. For example, for CustID: “C123”, the value is 1400, etc.



The following is the syntax that we will use to accomplish the objective:

```
db.Customers.mapReduce (
  map → function() { emit ( this.CustID, this.AccBal ); },
  reduce → function(key, values) { return Array.sum (values) },
  {
    query → query: { AccType: "S" },
    output → out: "Customer_Totals"
  }
)
```

## Map Function

```
var map = function(){
  emit ( this.CustID, this.AccBal );}
```

## Reduce Function

```
var reduce = function(key, values){ return Array.sum(values) ; }
```

```
C:\windows\system32\cmd.exe - mongo
> var reduce = function(key, values){ return Array.sum(values) ; }
>
```

## Executing the Query

```
db.Customers.mapReduce(map,      reduce, {out:      "Customer_Totals",      query:
{AccType:"S"}));
```

```
C:\windows\system32\cmd.exe - mongo
> db.Customers.mapReduce(map, reduce, {out: "Customer_Totals", query:{AccType:"S"}});
{
  "result" : "Customer_Totals",
  "timeMillis" : 7,
  "counts" : {
    "input" : 3,
    "emit" : 3,
    "reduce" : 1,
    "output" : 2
  },
  "ok" : 1,
}
>
```

The output as archived in “Customer\_Totals” collection:

```
C:\windows\system32\cmd.exe - mongo
> db.Customer_Totals.find().pretty();
{
  "_id" : "C111", "value" : 1200
}
{
  "_id" : "C123", "value" : 1400
}
>
```

## JavaScript Programming

**Objective:** To compute the factorial of a given positive number. The user is required to create a function by the name “factorial” and insert it into the “system.js” collection.

Before we proceed, let us do a quick check on what is contained in the “system.js” collection:

```
C:\windows\system32\cmd.exe - mongo
> db.system.js.find();
>
```

As per the screenshot above, currently there are no functions in the system.js collection.

*Act:*

```
db.system.js.insert({_id:"factorial",
value:function(n)
{
    if (n==1)
        return 1;
    else
        return n * factorial(n-1);
}
})
);
```

```
C:\windows\system32\cmd.exe - mongo
> db.system.js.insert({_id:"factorial",
... value:function(n)
... {
...     if (n==1)
...         return 1;
...     else
...         return n * factorial(n-1);
...     }
... });
> writeResult({ "nInserted" : 1 })
>
```

Confirm the presence of the “factorial” function in the system.js collection.

```
C:\windows\system32\cmd.exe - mongo
> db.system.js.find();
{ "_id" : "factorial", "value" : function (n)
{
    if (n==1)
        return 1;
    else
        return n * factorial(n-1);
} }
```

To execute the function, “factorial”, use the eval() method.

```
db.eval("factorial(3)");
```

```
C:\windows\system32\cmd.exe - mongo
> db.eval("factorial(3)");
6
>

db.eval("factorial(5)");

C:\windows\system32\cmd.exe - mongo
> db.eval("factorial(5)");
120
>

db.eval("factorial(1)");

C:\windows\system32\cmd.exe - mongo
> db.eval("factorial(1)");
1
>
```

#### 4.4.7 Cursors in MongoDB

**Objective:** To create a collection by the name “alphabets” and insert documents in it containing two fields “\_id” and “alphabet”. The values stored in the “alphabet” field should be “a”, “b”, “c”, “d”, etc. with one value stored per document. There should be 26 documents in all. We need to use cursor to iterate through the “alphabets” collection.

**Note:** “Alphabets” is the name of the collection and “alphabet” is the name of the field.

**Act:**

To create the collection “alphabets” with its 26 documents.

```
db.alphabets.insert({_id:1,alphabet:"a"});
db.alphabets.insert({_id:2,alphabet:"b"});
db.alphabets.insert({_id:3,alphabet:"c"});
db.alphabets.insert({_id:4,alphabet:"d"});
db.alphabets.insert({_id:5,alphabet:"e"});
db.alphabets.insert({_id:6,alphabet:"f"});
db.alphabets.insert({_id:7,alphabet:"g"});
db.alphabets.insert({_id:8,alphabet:"h"});
db.alphabets.insert({_id:9,alphabet:"i"});
db.alphabets.insert({_id:10,alphabet:"j"});
db.alphabets.insert({_id:11,alphabet:"k"});
db.alphabets.insert({_id:12,alphabet:"l"});
db.alphabets.insert({_id:13,alphabet:"m"});
db.alphabets.insert({_id:14,alphabet:"n"});
db.alphabets.insert({_id:15,alphabet:"o"});
db.alphabets.insert({_id:16,alphabet:"p"});
db.alphabets.insert({_id:17,alphabet:"q"});
db.alphabets.insert({_id:18,alphabet:"r"});
db.alphabets.insert({_id:19,alphabet:"s"});
db.alphabets.insert({_id:20,alphabet:"t"});
db.alphabets.insert({_id:21,alphabet:"u"});
db.alphabets.insert({_id:22,alphabet:"v"});
db.alphabets.insert({_id:23,alphabet:"w"});
db.alphabets.insert({_id:24,alphabet:"x"});
db.alphabets.insert({_id:25,alphabet:"y"});
db.alphabets.insert({_id:26,alphabet:"z"});
```

```
C:\windows\system32\cmd.exe - mongo
> db.alphabets.insert({_id:1,alphabet:"a"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:2,alphabet:"b"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:3,alphabet:"c"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:4,alphabet:"d"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:5,alphabet:"e"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:6,alphabet:"f"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:7,alphabet:"g"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:8,alphabet:"h"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:9,alphabet:"i"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:10,alphabet:"j"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:11,alphabet:"k"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:12,alphabet:"l"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:13,alphabet:"m"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:14,alphabet:"n"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:15,alphabet:"o"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:16,alphabet:"p"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:17,alphabet:"q"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:18,alphabet:"r"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:19,alphabet:"s"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:21,alphabet:"u"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:22,alphabet:"v"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:23,alphabet:"w"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:24,alphabet:"x"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:25,alphabet:"y"});
WriteResult({ "nInserted" : 1 })
> db.alphabets.insert({_id:26,alphabet:"z"});
WriteResult({ "nInserted" : 1 })
>
```

We now confirm the presence of 26 documents in the “alphabets” collection.

```
C:\windows\system32\cmd.exe - mongo
> db.alphabets.find();
{
  "_id" : 1, "alphabet" : "a"
}
{
  "_id" : 2, "alphabet" : "b"
}
{
  "_id" : 3, "alphabet" : "c"
}
{
  "_id" : 4, "alphabet" : "d"
}
{
  "_id" : 5, "alphabet" : "e"
}
{
  "_id" : 6, "alphabet" : "f"
}
{
  "_id" : 7, "alphabet" : "g"
}
{
  "_id" : 8, "alphabet" : "h"
}
{
  "_id" : 9, "alphabet" : "i"
}
{
  "_id" : 10, "alphabet" : "j"
}
{
  "_id" : 11, "alphabet" : "k"
}
{
  "_id" : 12, "alphabet" : "l"
}
{
  "_id" : 13, "alphabet" : "m"
}
{
  "_id" : 14, "alphabet" : "n"
}
{
  "_id" : 15, "alphabet" : "o"
}
{
  "_id" : 16, "alphabet" : "p"
}
{
  "_id" : 17, "alphabet" : "q"
}
{
  "_id" : 18, "alphabet" : "r"
}
{
  "_id" : 19, "alphabet" : "s"
}
{
  "_id" : 20, "alphabet" : "t"
}
Type "it" for more
>
```

A quick word on how the db.collection.find() method works. This is the primary method for read operation. In other words, it allows one to fetch the documents from the collection. To be able to access the documents, one needs to iterate the cursor.

However, in the mongo shell, if the returned cursor is not assigned to a variable using the var keyword, then the cursor is automatically iterated up to 20 times to print the first 20 documents in the result.

```
C:\windows\system32\cmd.exe - mongo
> var myCursor = db.alphabets.find();
> myCursor;
[{"_id": 1, "alphabet": "a"}, {"_id": 2, "alphabet": "b"}, {"_id": 3, "alphabet": "c"}, {"_id": 4, "alphabet": "d"}, {"_id": 5, "alphabet": "e"}, {"_id": 6, "alphabet": "f"}, {"_id": 7, "alphabet": "g"}, {"_id": 8, "alphabet": "h"}, {"_id": 9, "alphabet": "i"}, {"_id": 10, "alphabet": "j"}, {"_id": 11, "alphabet": "k"}, {"_id": 12, "alphabet": "l"}, {"_id": 13, "alphabet": "m"}, {"_id": 14, "alphabet": "n"}, {"_id": 15, "alphabet": "o"}, {"_id": 16, "alphabet": "p"}, {"_id": 17, "alphabet": "q"}, {"_id": 18, "alphabet": "r"}, {"_id": 19, "alphabet": "s"}, {"_id": 20, "alphabet": "t"}]
> Type "it" for more
```

Let us now look at designing manual cursors to iterate through the documents in the “alphabets” collection. We will use two methods with manual cursors: `hasNext()` and `next()`. A quick explanation of the two methods is as follows:

***hasNext() method:***

Return value: Boolean

The `hasNext()` method returns true if the cursor returned by the `db.Collection.find()` query can iterate further to return more documents.

***next() method:***

The `next()` method returns the next document in the cursor as returned by the `db.collection.find()` method.

```
C:\windows\system32\cmd.exe - mongo
> var myCur=db.alphabets.find({});
> while(myCur.hasNext()){
... var myRec=myCur.next();
... print("The alphabet is : " + myRec.alphabet);
...
The alphabet is : a
The alphabet is : b
The alphabet is : c
The alphabet is : d
The alphabet is : e
The alphabet is : f
The alphabet is : g
The alphabet is : h
The alphabet is : i
The alphabet is : j
The alphabet is : k
The alphabet is : l
The alphabet is : m
The alphabet is : n
The alphabet is : o
The alphabet is : p
The alphabet is : q
The alphabet is : r
The alphabet is : s
The alphabet is : t
The alphabet is : u
The alphabet is : v
The alphabet is : w
The alphabet is : x
The alphabet is : y
The alphabet is : z
>
```

The same result can be obtained by iterating through the cursor using a forEach loop.

```
C:\windows\system32\cmd.exe - mongo
> var cur=db.alphabets.find({});
> var myRec;
> cur.forEach( function(myRec) {
... print("The alphabet is : " + myRec.alphabet);
...
...
The alphabet is : a
The alphabet is : b
The alphabet is : c
The alphabet is : d
The alphabet is : e
The alphabet is : f
The alphabet is : g
The alphabet is : h
The alphabet is : i
The alphabet is : j
The alphabet is : k
The alphabet is : l
The alphabet is : m
The alphabet is : n
The alphabet is : o
The alphabet is : p
The alphabet is : q
The alphabet is : r
The alphabet is : s
The alphabet is : t
The alphabet is : u
The alphabet is : v
The alphabet is : w
The alphabet is : x
The alphabet is : y
The alphabet is : z
>
```

#### 4.4.8 Indexes

Assume the collection with the following documents:

```
> db.books.find().pretty();
{
  "_id" : 6,
  "Category" : "Machine Learning",
  "Bookname" : "Machine Learning for Hackers",
  "Author" : "Drew Conway",
  "qty" : 25,
  "price" : 400,
  "rol" : 30,
  "pages" : 350
}
{
  "_id" : 7,
  "Category" : "Web Mining",
  "Bookname" : "Mining the Social Web",
  "Author" : "Matthew A.Russell",
  "qty" : 55,
  "price" : 500,
  "rol" : 30,
  "pages" : 250
}
{
  "_id" : 8,
  "Category" : "Python",
  "Bookname" : "Python for Data Analysis",
  "Author" : "Wes McKinney",
  "qty" : 8,
  "price" : 150,
  "rol" : 20,
  "pages" : 150
}
{
  "_id" : 9,
  "Category" : "Visualization",
  "Bookname" : "Visualizing Data",
  "Author" : "Ben Fry",
  "qty" : 12,
  "price" : 325,
  "rol" : 6,
  "pages" : 450
}
{
  "_id" : 10,
  "Category" : "Web Mining",
  "Bookname" : "Algorithms for the intelligent web",
  "Author" : "Haralambos Marmanis",
  "qty" : 5,
  "price" : 850,
  "rol" : 10,
  "pages" : 120
}
>
```

Create an index on the key “Category” in the “books” collection.

```
> db.books.ensureIndex({"Category":1});
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Check on the status, that is, no. and name of the indexes:

```
> db.books.stats();
{
  "ns" : "test.books",
  "count" : 5,
  "size" : 1200,
  "avgObjSize" : 240,
  "storageSize" : 8192,
  "numExtents" : 1,
  "nindexes" : 2,
  "lastExtentSize" : 8192,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 1,
  "totalIndexSize" : 16352,
  "indexSizes" : {
    "_id_" : 8176,
    "Category_1" : 8176
  },
  "ok" : 1
}
```

Get the list of all indexes on the “books” collection:

```
> db.books.getIndexes();
[  
  {  
    "v" : 1,  
    "key" : {  
      "_id" : 1  
    },  
    "name" : "_id_",
    "ns" : "test.books"  
  },  
  {  
    "v" : 1,  
    "key" : {  
      "Category" : 1  
    },  
    "name" : "Category_1",
    "ns" : "test.books"  
  }  
]
```

To use the index on “Category” in the “books” collection, use the hint method:

```
> db.books.find({"Category": "Web Mining"}).pretty().hint({"Category":1});  
{  
  "_id" : 7,  
  "Category" : "Web Mining",  
  "Bookname" : "Mining the Social Web",  
  "Author" : "Matthew A.Russell",  
  "qty" : 55,  
  "price" : 500,  
  "rol" : 30,  
  "pages" : 250  
}  
{  
  "_id" : 10,  
  "Category" : "Web Mining",  
  "Bookname" : "Algorithms for the intelligent web",  
  "Author" : "Haralambos Marmanis",  
  "qty" : 5,  
  "price" : 850,  
  "rol" : 10,  
  "pages" : 120  
}  
.
```

Check the explain plan to get a deeper understanding on the use of index.

```

> db.books.find({"Category":"Web Mining"}).pretty().hint({"Category":1}).explain();
{
    "cursor" : "BtreeCursor Category_1",
    "isMultiKey" : false,
    "n" : 2,
    "nscannedObjects" : 2,
    "nscanned" : 2,
    "nscannedObjectsAllPlans" : 2,
    "nscannedAllPlans" : 2,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "indexBounds" : {
        "Category" : [
            [
                "Web Mining",
                "Web Mining"
            ]
        ]
    },
    "server" : "PUNITP123103L:27017",
    "filterSet" : false
}

```

Let us look at the case of covered index. Observe that the “indexOnly” property will be set to true for covered index.

```

> db.books.find({"Category":"Web Mining"}, {"Category":1, "_id":0}).pretty().hint({"Category":1}).explain();
{
    "cursor" : "BtreeCursor Category_1",
    "isMultiKey" : false,
    "n" : 2,
    "nscannedObjects" : 0,
    "nscanned" : 2,
    "nscannedObjectsAllPlans" : 0,
    "nscannedAllPlans" : 2,
    "scanAndOrder" : false,
    "indexOnly" : true,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "indexBounds" : {
        "Category" : [
            [
                "Web Mining",
                "Web Mining"
            ]
        ]
    },
    "server" : "PUNITP123103L:27017",
    "filterSet" : false
}

```

In order to have the index cover the query, ensure that only those columns are projected on which the index is built. In the above example, the index is built on the “Category” column, and “Category” is the only column that is projected. Even the identifier (\_id) is suppressed.

#### 4.4.9 Mongolimport

This command is used when the command prompt imports CSV (Comma Separated Values) or TSV (Tab Separated Values) files or JSON (Java Script

Object notation) documents into MongoDB.

**Objective:** Given a CSV file “sample.txt” in the D: drive, import the file into the mongoDB collection, “SampleJSON”. The collection is in the database “test”.

The “sample.txt” file is as follows:

```
_id,FName,LName
1,Samuel,Jones
2,Virat,Kumar
3,Raul,"A Simpson"
4,"Andrew Simon"
```

**Act:** At the command prompt, execute the following command.

```
Mongoimport --db test --collection SampleJSON --type csv --headerline
--file d:\sample.txt
```

On successful execution of the command, the message at the prompt will be as follows:

```
|connected to: 127.0.0.1
|2015-02-20T21:09:27.301+0530 imported 4 objects
```

**Output:** To confirm the output, log into MongoDB shell, navigate to the “SampleJSON” collection in the “test” database.

The following are the JSON documents in the collection:

```

> db
test
> show collections
Customers
SampleJSON
books
fs.chunks
fs.files
persons
system.indexes
usercounters
users
> db.SampleJSON.find().pretty();
{
  "_id" : 1, "FName" : "Samuel", "LName" : "Jones" }
{
  "_id" : 2, "FName" : "Virat", "LName" : "Kumar" }
{
  "_id" : 3, "FName" : "Raul", "LName" : "A Simpson" }
{
  "_id" : 4, "FName" : "", "LName" : "Andrew Simon" }
>

```

### *Upsert with mongoimport*

```

> use sample
switched to db sample

> db
sample

> db.Persons.insert(
... {
...   "_id" : 1001,
...   "name" : "Nazia Jabeen",
...   "region" : "United States",
...   "email" : "nazia@example.com"
... });
writeResult({ "nInserted" : 1 })

> db.Persons.find();
{ "_id" : 1001, "name" : "Nazia Jabeen", "region" : "United States", "email" : "nazia@example.com" }

Persons_1001.json
{
  "_id" : 1001,
  "username" : "Nazia",
  "likes" : [ "running", "pandas", "software development" ]
}
mongoimport -c Persons -d sample --upsert --file D:/Persons_1001.json
--jsonArray

```

```
D:\BackupfromOldLaptop\2015\MongoDB\bin>mongoimport -c Persons -d sample --upsert --file d:/Persons_1001.json --jsonArray
connected to: 127.0.0.1
2019-03-04T22:26:29.377+0530 imported 1 objects
```

```
> db.Persons.find().pretty();
{
    "_id" : 1001,
    "username" : "Nazia",
    "likes" : [
        "running",
        "pandas",
        "software development"
    ]
}
```

---

## Merge

```
> db.Persons.find().pretty();
{
    "_id" : 1001,
    "name" : "Nazia Jabeen",
    "region" : "United States",
    "email" : "nazia@example.com"
}

Persons_1001
{
    "_id" : 1001,
    "username" : "Nazia",
    "likes" : [ "running", "pandas", "software development" ]
}

mongoimport -c Persons -d sample --merge --file D:/Persons_1001.json --jsonArray

{
    "_id": 1001,
    "name": "Nazia Jabeen",
    "region": "United States",
    "email": Nazia@example.com,
    "username": "Nazia",
    "likes": [ "running", "pandas", "software development" ]
}
```

## 4.4.10 MongoExport

This command used at the command prompt exports MongoDB JSON documents into CSV (Comma Separated Values) or TSV (Tab Separated Values) files or JSON (Java Script Object notation) documents.

**Objective:** This command used at the command prompt exports MongoDB JSON documents from “Customers” collection in the “test” database into a CSV file “Output.txt” in the D: drive.

Given below is a snapshot of the JSON documents in the “Customers” collection of the “test” database.

```
> db
test
> show collections
Customers
SampleJSON
books
fs.chunks
fs.files
persons
system.indexes
usercounters
users
> db.Customers.find().pretty();
{
  "_id" : ObjectId("54df6d4f46a31d28183b9a5b"),
  "CustID" : "C123",
  "AccBal" : 500,
  "AccType" : "S"
}
{
  "_id" : ObjectId("54df6d4f46a31d28183b9a5c"),
  "CustID" : "C123",
  "AccBal" : 900,
  "AccType" : "S"
}
{
  "_id" : ObjectId("54df6d4f46a31d28183b9a5d"),
  "CustID" : "C111",
  "AccBal" : 1200,
  "AccType" : "S"
}
{
  "_id" : ObjectId("54df6d4f46a31d28183b9a5e"),
  "CustID" : "C123",
  "AccBal" : 1500,
  "AccType" : "C"
}
>
```

**Act:** At the command prompt, execute the following command:

```
Mongoexport --db test --collection Customers --csv --fieldFile d:\fields.txt --out d:\output.txt
```

Before executing this command, ensure that you created a “fields.txt” with a format defined as follows:

The “fields.txt” file:

```
CustID
AccBal
AccType
```

For the MongoExport command to execute successfully, ensure that the fields are spelled as is in the MongoDB collection. The case also has to be maintained. It is mandatory to ensure that only one field name is placed per line.

On successful execution of the command, the message at the prompt will be as follows:

```
connected to: 127.0.0.1
exported 4 records
```

**Output:** To confirm the output, navigate to the D: drive and check the file “Output.txt”.

```
"Output.txt"
CustID,AccBal,AccType
"C123",500.0,"S"
"C123",900.0,"S"
"C111",1200.0,"S"
"C123",1500.0,"C"
```

---

#### 4.4.11 Automatic Generation of Unique Numbers for the “\_id” Field

**Step 1:** Run the insert() method on a new collection “usercounters”. This is to start off with an initial value of 0 for the “seq” field.

```
db.usercounters.insert(
{
  _id: "empid",
  seq:0
})
```

**Step 2:** Create a user-defined function “getnextseq”. This method will invoke “findAndModify()” method on the “usercounters” collection. This is to increment the value of seq field by 1 and update the same in “usercounters” collection.

```
function getnextseq(name) {  
  var ret=db.usercounters.findAndModify(  
  {  
    query: {_id:name},  
    update: {$inc:{seq:1}},  
    new:true  
  }  
);  
  return ret.seq;  
}
```

**Step 3:** Run the insert() method on the collection where you need to have the “\_id” field get the uniquely generated number. Note the call to getnextseq() method as value to \_id. The return value from the getnextseq() method becomes the value of \_id.

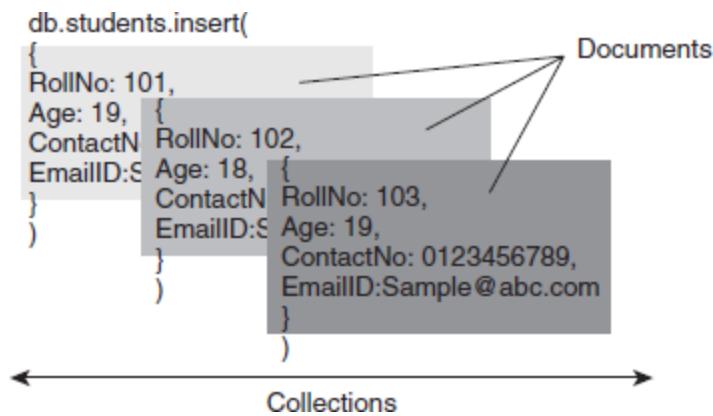
```
db.users.insert(  
{  
  _id:getnextseq("empid"),  
  Name: "sarah jane"  
})
```

## REMEMBER ME



- 
1. MongoDB is:
    - (a) Cross-platform
    - (b) Open source
    - (c) Non-relational
    - (d) Distributed
    - (e) NoSQL
    - (f) Document-oriented data store
  2. MongoDB is a collection of documents. Each document is a collection of key-value pairs. A document is analogous to a row/record/tuple in an RDBMS table. A document has a dynamic schema. This implies that a

document in a collection need not necessarily have the same set of fields/key-value pairs.



3. JSON is extremely expressive. MongoDB actually does not use JSON but BSON (pronounced Bee Son), which is binary JSON. It is an open standard, and it is used to store complex data structures.
4. A collection in MongoDB is analogous to a table of RDBMS. A collection does not enforce a schema. This implies that documents within a collection can have different fields. Even if the documents within a collection have same fields, the order of the fields can be different.
5. Sharding is akin to horizontal scaling. It means the large dataset is divided and distributed over multiple servers or shards. Each shard is an independent database and collectively they would constitute a logical database.
6. MongoDB updates the information in-place. This implies that it updates the data wherever it is available. It does not allocate separate space and the indexes remain unaltered.

## TEST ME

---

1. MongoDB supports dynamic schema design.
  - (a) False
  - (b) True
2. MongoDB supports query joins between collections.
  - (a) False
  - (b) True

3. Which of the following MongoDB conditional operator is not a valid operator?

- (a) \$lt
- (b) \$ltu
- (c) \$gt
- (d) \$lte

4. What does the following command do?

```
db.demo.find({ "extra.community_name" : "Rock", "extra.friends.valued_friends_id": "Jhon" })
```

- (a) Fetch documents from the collection "demo" wherein the embedded document "extra" contains the key field "community\_name" with value "Rock" and the embedded document "extra" contains another embedded document "friends" with key field "valued\_friends\_id" which contains the value "Jhon".
- (b) Fetch documents from the collection "demo" which contain key field "community\_name" with value "Rock" and key field "valued\_friends\_id" with value as "Jhon".

5. '\$unset' is used with

- (a) Insert
- (b) Update
- (c) Delete

6. MongoDB is supported by

- (a) Perl
- (b) Python
- (c) PHP
- (d) All the above.

7. MongoDB is

- (a) RDBMS
- (b) Object-oriented DBMS
- (c) Document-oriented DBMS
- (d) Key-value store

8. 'mongoimport' command is used

- (a) for multiple command insertion

- (b) to provide a route to import content from a JSON, CSV, or TSV export created by mongoexport,  
(c) for multiple command import
9. Which of the following is the correct command to insert data into MongoDB?
- Assume that document is a valid JSON document.
- (a) db.Students.insert(document)  
(b) db.Students.insert().(document)  
(c) Students.insert(document)
10. MongoDB documents are represented as
- (a) XML  
(b) JSON  
(c) DOCUMENT
11. MongoDB supports unique indexes just like most other relational database.
- (a) True  
(b) False
12. Which of the following command creates an index, where mobile\_no is a field in the collection employees?
- (a) db.employees.SetIndex( { "mobile\_no": 1 } )  
(b) db.employees.ensureIndex("mobile\_no": 1 )  
(c) employees.SetIndex( { "mobile\_no": 1 } )
13. Which of the following command is correct when you want to fetch documents from collection only employees whose salary is either 8500 or 10,000?
- (a) db.employees.find.sort({“salary”:\$in:[8500,10000]})  
(b) db.employees.find({“salary”:\$in:[8500,10000]})  
(c) db.employees.find({“salary”:\$in:[8500,10000]})
14. Which of the following is the command equivalent to
- Select first\_name,salary,date\_of\_join from employees where designation=“Manager”;
- (a) db.employees.find({“designation”:“Manager”},{“first\_name” : 1;“salary”:1;“date\_of\_join”:1})  
(b) db.employees.find({“designation:Manager”},{“first\_name” : 1,“salary”:1,“date\_of\_join”:1})

(c) db.employees.find({“designation”:“Manager”},{“first\_name” : 1,“salary”:1,“date\_of\_join”:1})

15. MongoDB enforces attribute similarity across documents in a collection

- (a) True
- (b) False

16. The maximum BSON document size is

- (a) 8 megabytes
- (b) 4 megabytes
- (c) 32 megabytes
- (d) 16 megabytes

17. Core MongoDB Operations are

- (a) Create, Select, Update, Delete
- (b) Create, Read, Update, Delete
- (c) Create, Read, Update, Drop

18. Which of the following command provides you with a list of all the databases in MongoDB?

- (a) show databases
- (b) show dbs
- (c) show all dbs
- (d) None of the above

19. If we want to remove the document from the collection ‘employees’ which contains the ‘first\_name’ as “John” the following mongodb command can be used:

- (a) db.userdetails.remove({})
- (b) db.employees.remove( { “first\_name : John” } )
- (c) db.employees.remove( { “first\_name” : “John” } )

20. What does the following command do?

db.sample.find().limit(10)

- (a) Shows 10 documents randomly from the collection sample
- (b) Shows only first 10 documents from the collection sample
- (c) Repeats the first document 10 times

21. Which one of the following is equivalent to?

Select \* from employees order by salary

- (a) db.employees.find().sort({“salary:1”})

- (b) db.employees.sort({“salary”:1})  
(c) db.employees.find().sort({“salary”:1})
22. Which command in MongoDB is equivalent to SQL select?
- (a) search()  
(b) find()  
(c) document()
23. Which of the following is equivalent to the following?  
select first\_name,salary from employees where designation=“Manager”;  
Assume that there are three columns first\_name, salary, date\_of\_join.
- (a) db.employees.find({“designation:Manager”},{“date\_of\_join” : 0})  
(b) db.employees.find({“designation:Manager”},{“date\_of\_join” : 1})  
(c) db.employees.find({“designation”:“Manager”},{“date\_of\_join” : 0})
24. Which of the following answers equals to SQL command – Select \* from employees where designation=“Manager”;
- (a) employees.find({“designation”:“manager”})  
(b) db.employees.find({“designation:manager”})  
(c) db.employees.find({“designation”:“manager”})
25. Which of the following is correct command to update?
- (a) db.books.update( { item: “book”, qty: { \$gt: 7 } }, { \$set: { x: 5 }, \$inc: { y: 8} } )  
(b) db.books.find().update( { item: “book”, qty: { \$gt: 7 } }, { \$set: { x: 5 }, \$inc: { y: 8} } )  
(c) db.books.update( { item: “book”, qty: { \$gt: 7 } }, { \$set: { x: 5 }, \$inc: { y: 8} } , { multi: true } )
26. Which one of the following is equivalent to  
Select \* from employees order by salary desc;
- (a) db.employees.find().sort({“salary”:1})  
(b) db.employees.find().sort({“salary”:-1})  
(c) db.employees.sort({“salary”:-1})
27. Which of the following command is correct when you want to fetch documents from collection where only employees whose either salary is 7500 or date of join is 17/10/2009 would come?
- (a) db.employees.find({ “salary” : “7500” , (\$or : [ { “date\_of\_join” : “17/10/2009” } ] } ))

- (b) db.employees.find({ "salary" : "7500" , \$or : [ { "date\_of\_join" : "17/10/2009" } ] })
- (c) db.employees.find().sort({ "salary" : "7500" , \$or : [ { "date\_of\_join" : "17/10/2009" } ] })

**28.** What does the following command do?

```
db.employees.find().skip(5).limit(5)
```

- (a) Skips first five documents and then shows just next five documents
- (b) Shows just next five documents
- (c) Skips first five documents and then shows the sixth document five times

**29.** Which of the following command is correct when you want to fetch documents from collection demo, where value of a field 'interest' is null?

- (a) db.demo.find( { "interest" : null } )
- (b) db.demo.find().sort( { "interest" : null } )
- (c) db.demo.find( { "interest : null" } )

**30.** Which one of the following answers is equivalent to this SQL command?

Select \* from employees where date\_of\_join="16/10/2010" and designation="Manager" order by salary desc;

- (a) db.employees.find({“date\_of\_joinä:  
16/10/2010”,“designation”：“Manager”}).sort({“salary:-1”})
- (b) db.employees.find().sort({“date\_of\_join”:  
“16/10/2010”,“designation”：“Manager”}).sort({“salary”:-1})
- (c) db.employees.find({“date\_of\_join”:  
“16/10/2010”,“designation”：“Manager”}).sort({“salary”:-1})

## QUESTION ME

---

1. What is MongoDB? Explain.
2. What is sharding in MongoDB?
3. How does sharding work with replication?
4. What is a collection in MongoDB?
5. What is the syntax to create and drop a collection in MongoDB?
6. Explain indexes in MongoDB.
7. What is aggregation in MongoDB?

8. What is embedded document in MongoDB?
9. What is the use of pretty() method?
10. Define MapReduce in MongoDB.

## REFERENCE ME

---

1. <https://www.mongodb.com/>
2. <https://en.wikipedia.org/wiki/MongoDB>
3. <https://searchdatamanagement.techtarget.com/definition/MongoDB>
4. <https://docs.mongodb.com/manual/tutorial/>

## ANSWERS

---

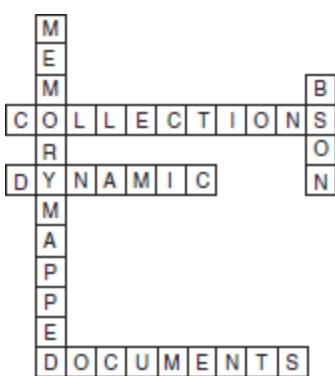
### Try This – 1

1. MongoDB
2. MongoDB
3. Traditional RDBMS
4. CouchDB

### Try This – 2

(b) and (d)

### Try This – 3



## Try This – 4

```
db.Person.find()  
[{"_id": 3, "Name": "Jeet Acharya"},  
 {"_id": 4, "Name": "Sanjeet Acharya"}]
```

## Try This – 5

1. (b), (c)
2. (c)
3. (a), (d)

## Test Me

1. (b)
2. (a)
3. (b)
4. (a)
5. (b)
6. (d)
7. (c)
8. (b)
9. (a)
10. (b)
11. (a)
12. (b)
13. (c)
14. (c)
15. (b)
16. (d)
17. (b)
18. (b)
19. (c)
20. (b)
21. (c)
22. (b)
23. (c)
24. (c)

**25.** (a)

**26.** (b)

**27.** (b)

**28.** (a)

**29.** (a)

**30.** (c)

*OceanofPDF.com*



# Neo4j: A Graph-Based Database

---

## BRIEF CONTENTS

- Introduction to Graph Database
- Creating Nodes
  - Create a Single Node
  - Verify the Creation of the Node
  - Create Multiple Nodes
  - Create a Node with a Label
  - Create a Node with Multiple Labels
  - Create a Node with Properties
  - Return a Created Node
- Create a Relationship
  - Create a Relationship with Label and Properties
- WHERE Clause
  - WHERE Clause with Multiple Conditions
  - COUNT() Function
- Creating a Complete Path
- Create Index
- Create Constraints

- Select Data with MATCH
- Fetch all Nodes
- Drop an Index
- Drop a Constraint
- Delete a Node
- Delete Multiple Nodes
- Delete all Nodes
- Delete a Relationship
- Merge Command

## 5.1 INTRODUCTION TO GRAPH DATABASE

---

### Picture This

You have been asked to design a model for access control lists. There are users, the various roles that they play and then there are resources which are assigned to roles.

Think RDBMS (relational database management system) and you would need at least 5 relational tables to store the information. A table to store details about users. A table to store the roles. A table to store information about resources. Now comes the difficult part, how to map users to roles and roles to resources. You will need two, many-to-many mapping tables to store information about which users play which roles and which roles have been assigned which resources.

**Users table**

| EmployeeID | EmployeeName | EmployeeEmailID | Projectcode | ----- |
|------------|--------------|-----------------|-------------|-------|
| E101       | Alexa        | Alexa@abc.xyz   | P109        | ----- |

**Role table**

| RoleID | Role           | Description                                                                                       | AreaOfResponsibility                                                                                                                                                                                    |
|--------|----------------|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|        |                |                                                                                                   | --                                                                                                                                                                                                      |
|        |                |                                                                                                   | --                                                                                                                                                                                                      |
|        |                |                                                                                                   | -                                                                                                                                                                                                       |
| R1     | ProjectManager | <p>A Project Manager interfaces with the development team and the client. He/She</p> <p>.....</p> | <ul style="list-style-type: none"> <li>(a) Allocation/deallocation of employees from the projectcode</li> <li>(b) Single Point of Contact (SPOC for clients)</li> <li>(c) Etc...</li> </ul>             |
| R2     | DBAdmin        | <p>Data Base administrator. Controls access rights to the data base servers</p>                   | <ul style="list-style-type: none"> <li>(a) Assigns read/write privileges to project team members</li> <li>(b) Backups the database servers</li> <li>(c) Periodically cleans up the databases</li> </ul> |

**Resource table**

| ResourceID | ResourceName | ----- |
|------------|--------------|-------|
| R101       | DBServer01   | ----- |
| R102       | DBServer02   | ----- |
| R103       | Printer01    | --    |

**User role mapping table**

| RoleID | UserID | ValidFrom    | ValidTo     | ----- |
|--------|--------|--------------|-------------|-------|
| R1     | E101   | 01-June-2019 | 30-Sep-2019 | ----- |
| R2     | E101   | 01-June-2019 | 30-Sep-2019 | ----- |

Role resource mapping table

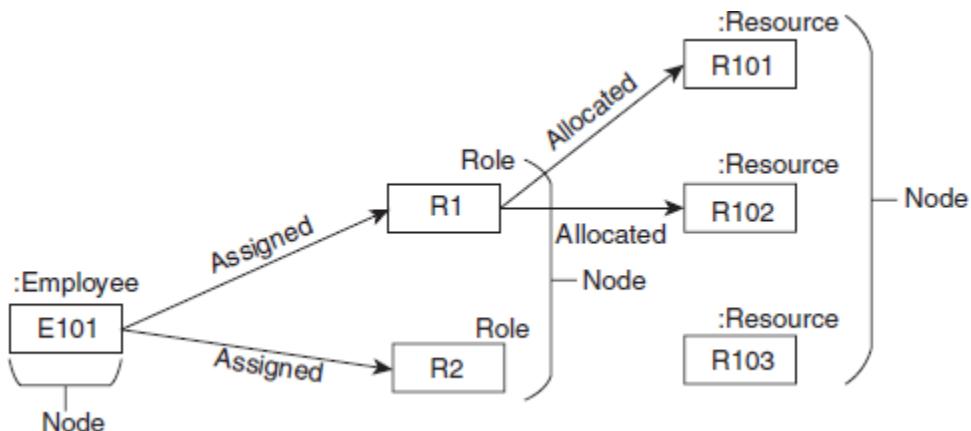
| RoleID | ResourceID | AccessPermission | Description    |
|--------|------------|------------------|----------------|
| R1     | R101       | RW               | Read and Write |
| R1     | R102       | RW               | Read and Write |

The users are the employees of the enterprise.

Wouldn't it be more convenient to represent data in its natural form, making mappings more intuitive and skipping the repeated process of translating data to and from a storage engine. Well, now you can. We have the graph databases which allow data to be stored as a graph with a structure consisting of nodes or vertices and edges or relationships. Refer [Figure 5.1](#).

### ***What is a graph?***

A graph is about nodes and relations between nodes.



**Figure 5.1** Sample graph diagram (Employee E101 is in Roles R1 and R2 and Role R1 is allocated resources R101 and R102).

### ***Why graph database?***

It is easier to represent relationships between nodes.

Let us look at a few typical use cases:

1. Access control lists
2. Recommendation engines
3. Social networks
4. Interconnected data

## Examples:

1. Neo4j
2. Allegrograph
3. OrientDB

Let us study Neo4j in detail starting with the creation of a single node.

## 5.2 CREATING NODES

---

### 5.2.1 Create a Single Node

Let us begin with creating our very first node.

Syntax to create a node is

```
CREATE (node_name);
```

**Example:** In order to create a node by the name “Single\_Node”, type the below command at the dollar prompt.

*Command:*

```
$ Create (Single_Node);
```

*Output:*

```
$ Create (Single_Node);
```

Table

Code

Created 1 node, completed after 1077 ms.

You will notice that nothing is returned from this query except for the count of affected nodes. The number of nodes created in the example above is “1”.

### 5.2.2 Verify the Creation of the Node

To check/verify whether the node “Single\_Node” was created successfully, use the below command:

**Command:**

```
$ MATCH (n) RETURN n;
```

The command returns all the nodes in the database.

**Output:**



```
$ MATCH (n) RETURN n ;
```

Graph

Table

Text

Code

Displaying 1 nodes, 0 relationships.

| Node        | Type | Properties |
|-------------|------|------------|
| Single_Node | Node | {} (empty) |

You can observe that a node has been created but with no relationships.

### 5.2.3 Create Multiple Nodes

To create multiple nodes, the syntax of the command is

```
CREATE (node1), (node2), (node1), (node n) ..
```

**Example:** To create two nodes “Primary\_Node” and “Secondary\_Node”, type in the below command at the dollar prompt.

**Command:**

```
$ Create (Primary_Node), (Secondary_Node);
```

```
$ CREATE (Primary_Node), (Secondary_Node);
```

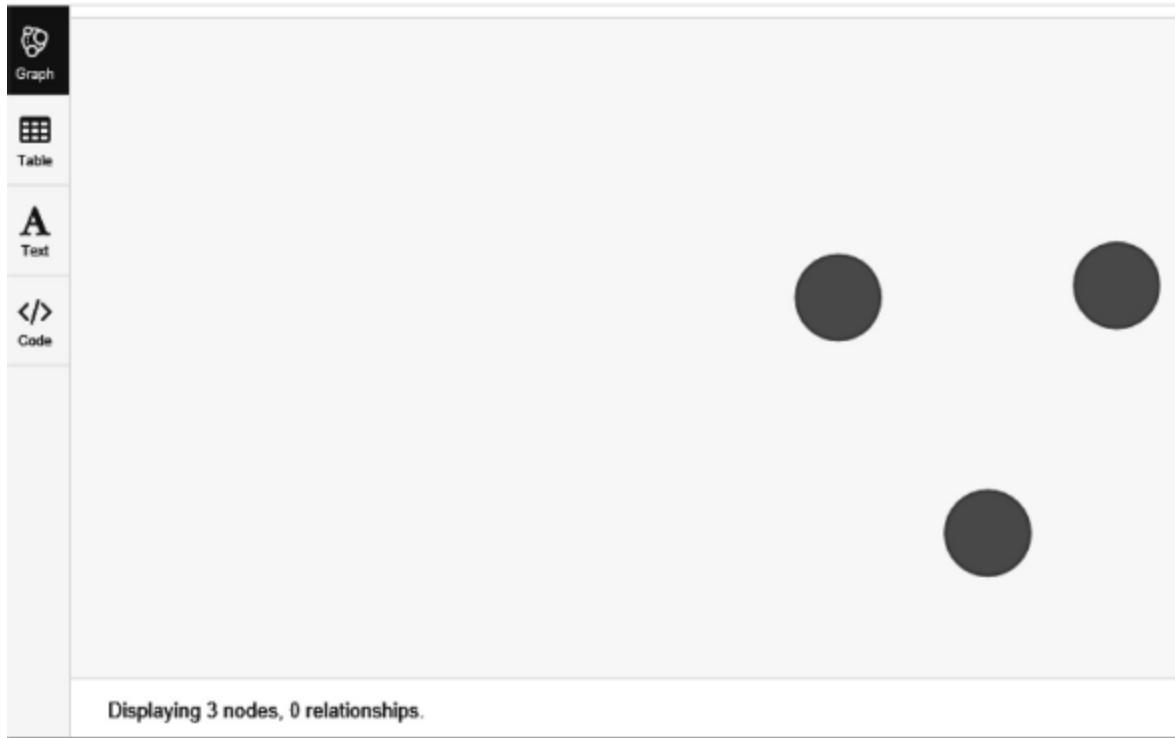
**Output:**

```
$ CREATE (Primary_Node), (Secondary_Node);  
Table  
Created 2 nodes, completed after 8 ms.  
</>  
Code
```

Two nodes have been created. To quickly verify the successful creation of the two nodes, type in the command:

```
$ MATCH (n) RETURN (n);
```

```
$ MATCH (n) RETURN n ;
```



The screenshot shows the Neo4j browser interface. On the left is a vertical sidebar with four items: 'Graph' (selected), 'Table', 'Text', and 'Code'. The main area displays three dark grey circular nodes arranged in a triangle. At the bottom of the main area, the text 'Displaying 3 nodes, 0 relationships.' is visible.

It shows a total of three nodes “Single\_Node”, “Primary\_Node”, and “Secondary\_Node”.

#### 5.2.4 Create a Node with a Label

To create a node with a label, the syntax is

```
CREATE (node:label)
```

**Command:**

```
$ CREATE (`Albert Einstein`: physicist)
```

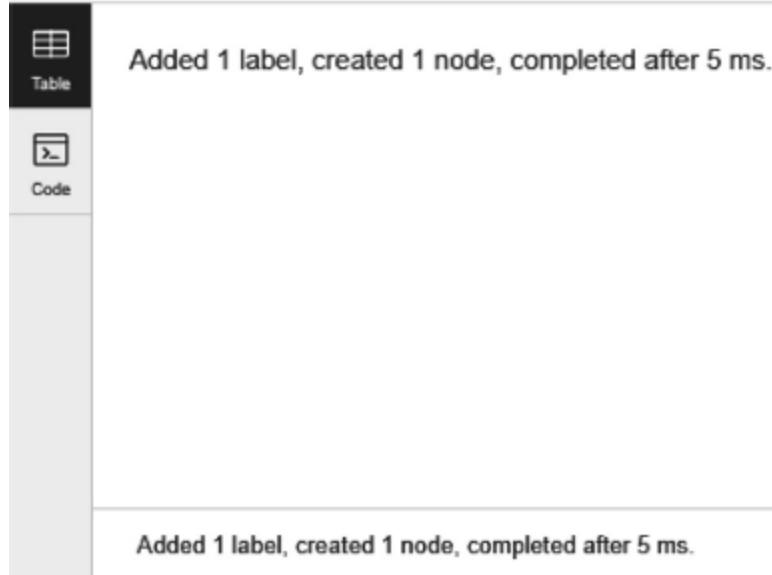
Here the node is “Albert Einstein” and the label is “physicist”.

**Output:**

---

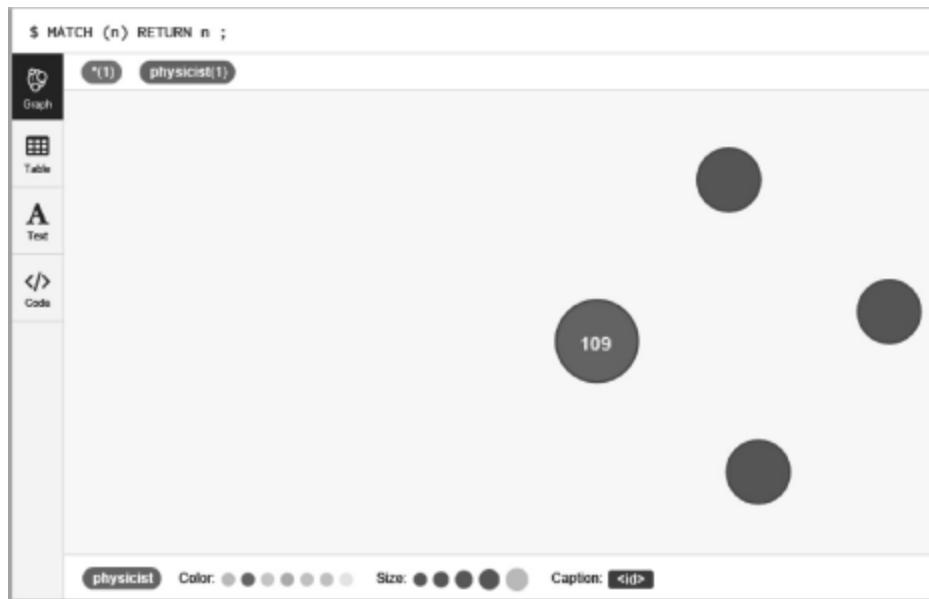
```
$ CREATE (`Albert Einstein`: physicist)
```

---



A node has been created with a label “physicist”.

**Verification:** Verify that the node is successfully created using match command.



## 5.2.5 Create a Node with Multiple Labels

*Syntax:*

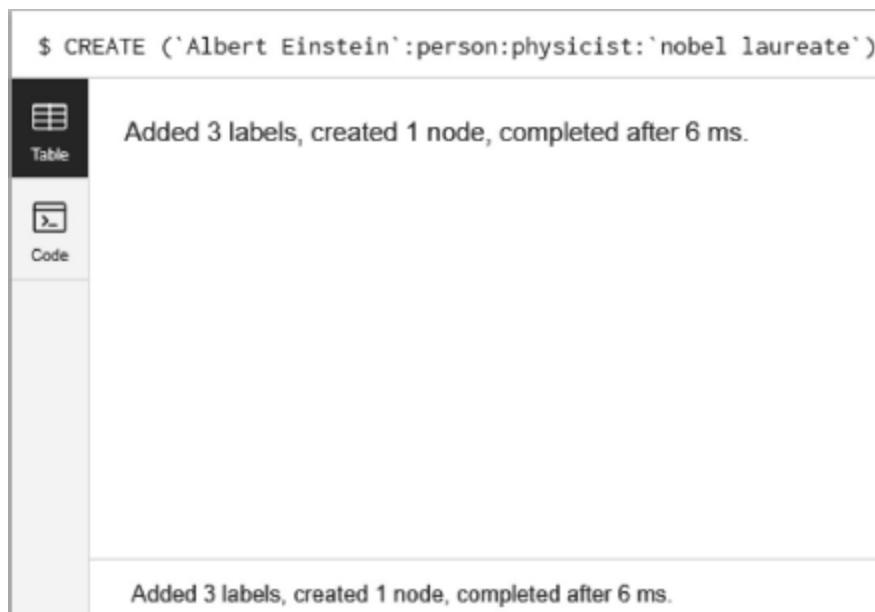
```
CREATE (node:label1:label2:...:labeln)
```

**Example:** To create a node “Albert Einstein” with labels “person”, “physicist”, and “nobel laureate”, the following command is used:

**Command:**

```
$ CREATE ('Albert Einstein':person:physicist:'nobel laureate')
```

**Output:**



The image shows a terminal window with the following content:

```
$ CREATE ('Albert Einstein':person:physicist:'nobel laureate')
```

Added 3 labels, created 1 node, completed after 6 ms.

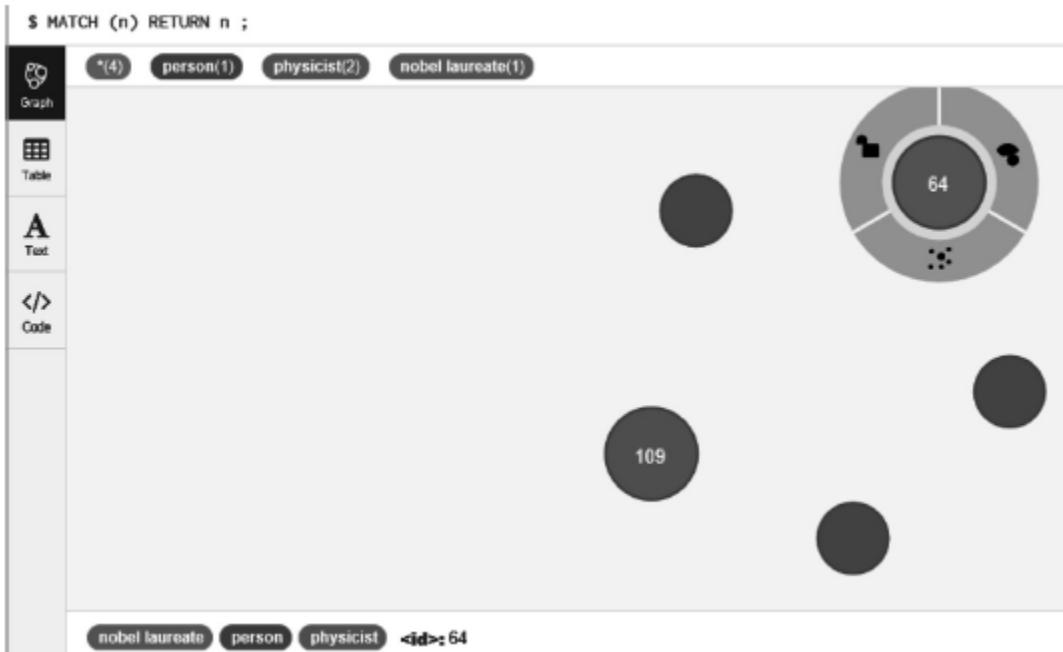
Code

---

```
Added 3 labels, created 1 node, completed after 6 ms.
```

**Note:** A node has been created with three labels “person”, “physicist” and “nobel laureate”.

**Verification:** A quick verification of the node created with the three labels (person, physicist, and nobel laureate) using the match command.



## 5.2.6 Create a Node with Properties

*Syntax:*

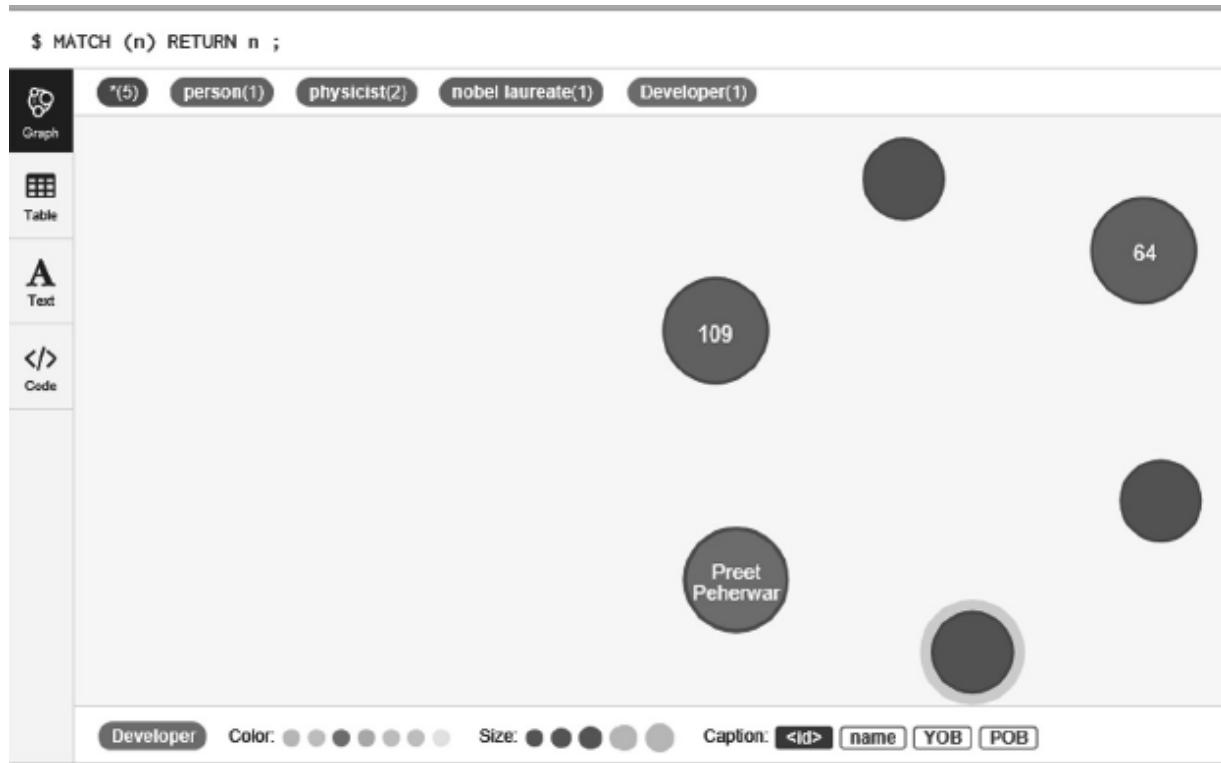
```
$ CREATE (node:label { key1: value, key2: value, . . . . . . . . . . . . })
```

**Example:** To create a node “Preet Peherwar” with label “Developer” and properties which are specified as key value pairs (name: “Preet Peherwar”, YOB: 1983 and POB: “Mumbai”) use the below command. Here YOB is “Year Of Birth” and POB is “Place Of Birth”.

*Command:*

```
$ CREATE ('Preet Peherwar':Developer{name: "Preet Peherwar", YOB: 1983, POB: "Mumbai"})
```

*Output:*



### 5.2.7 Return a Created Node

*Syntax:*

```
CREATE (Node:Label{properties. . . . }) RETURN Node
```

**Example:** Use the below command to create a node “Kiran” with label “Evangelist” and properties (name: “Kiran Radhakrishan”, YOB: 1971 and POB: “Karnataka”) and after creating the node return it.

*Command:*

```
$ CREATE (Kiran:Evangelist{name: "Kiran Radhakrishan", YOB: 1971, POB: "Karnataka"}) RETURN Kiran
```

*Output:*



As can be seen in the above screen shot, the node “Kiran Radhakrishan” is created and also returned without having to use Match(n) Return(n) command.

## 5.3 CREATE A RELATIONSHIP

---

To create a relationship between nodes, the syntax is

```
CREATE (node1)-[:RelationshipType]->(node2)
```

### Example:

**Step 1:** Create a node “Lionel” with label “player” and properties (name: “Lionel Messi”, YOB: 1987, POB: “Argentina”).

```
$ CREATE (Lionel:player{name: "Lionel Messi", YOB: 1987, POB: "Argentina"})
```

**Step 2:** Create a node “Team” with label “CurrentTeam” and properties (name: “Barcelona”).

```
$ CREATE (Team:CurrentTeam {name: "Barcelona"})
```

**Step 3:** Return the nodes Lionel and Team.

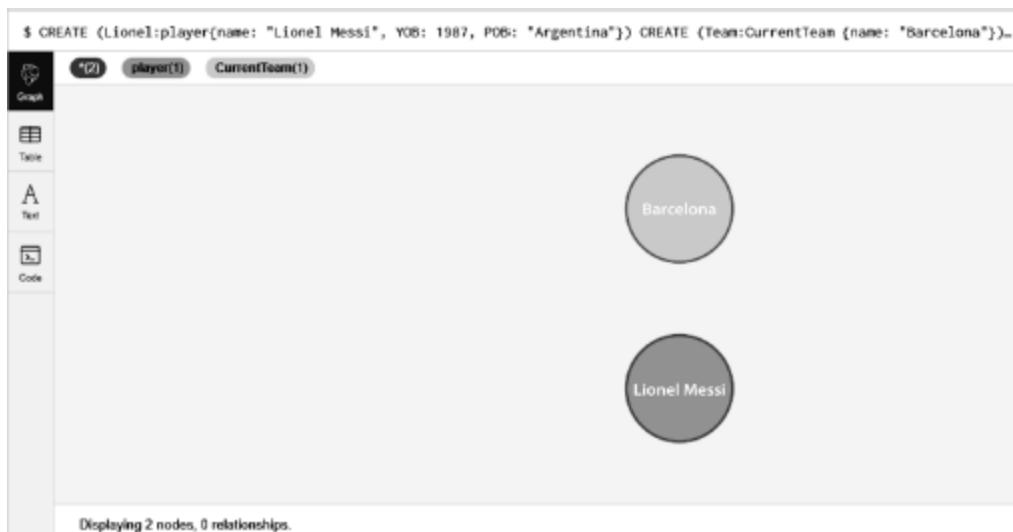
```
RETURN Lionel, Team
```

**Note:** Bundle all these three statements and execute them together. Do not use a semicolon at the end of every statement.

**Input:**

```
1 CREATE (Lionel:player{name: "Lionel Messi", YOB: 1987, POB: "Argentina"})
2 CREATE (Team:CurrentTeam {name: "Barcelona"})
3 RETURN Lionel, Team
```

**Output:**



As is obvious from the above screen shot, two nodes are created with labels “player” and “CurrentTeam”.

**Step 4:** Create a Relationship between existing Nodes

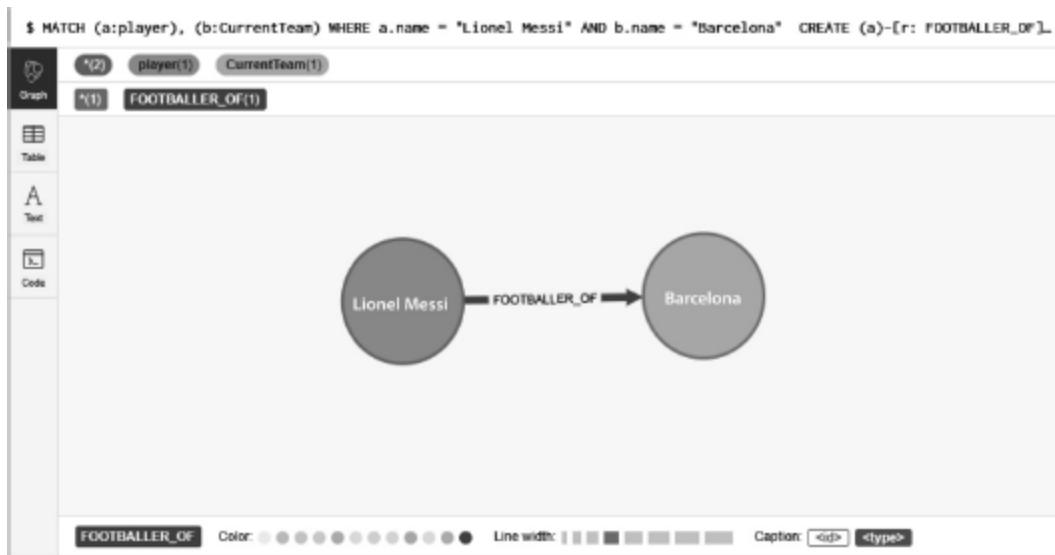
**Command:**

```
1 MATCH (a:player), (b:CurrentTeam) WHERE a.name = "Lionel Messi" AND b.name = "Barcelona"
2 CREATE (a)-[r: FOOTBALLER_OF]->(b)
3 RETURN a,b
```

In the above example, node with label “player” is denoted as “a” and node with label “CurrentTeam” is denoted as “b”. A relationship “FOOTBALLER\_OF”

is created between a and b. In other words, relationship “FOOTBALLER\_OF” is created between the nodes with labels “player” and “CurrentTeam”.

### ***Output:***



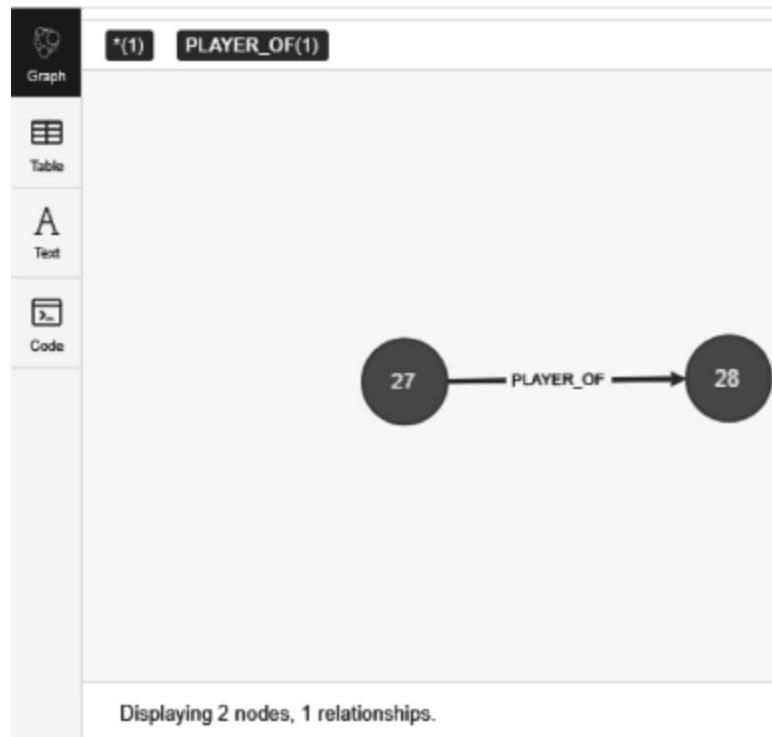
Now let us create a relationship “PLAYER\_OF” between the two nodes “Lionel” and “Team”.

### ***Command:***

```
1 CREATE (Lionel)-[r:PLAYER_OF]->(Team)
2 RETURN Lionel,Team
```

### ***Output:***

```
$ CREATE (Lionel)-[r:PLAYER_OF]->(Team) RETURN Lionel, Team
```



The output can also be viewed in tabular or text form.

### *Output in tabular form:*

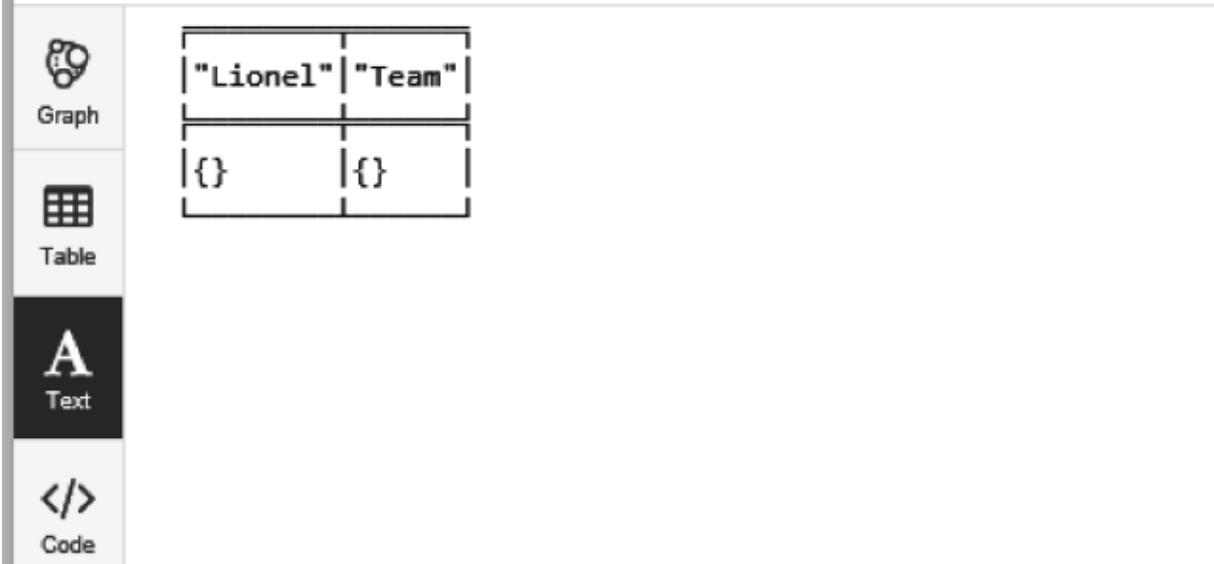
```
$ CREATE (Lionel)-[r:PLAYER_OF]->(Team) RETURN Lionel, Team
```

|  | Lionel | Team   |
|--|--------|--------|
|  | {<br>} | {<br>} |

Created 2 nodes, created 1 relationship, started streaming 1 records after 6 ms and completed after 7 ms.

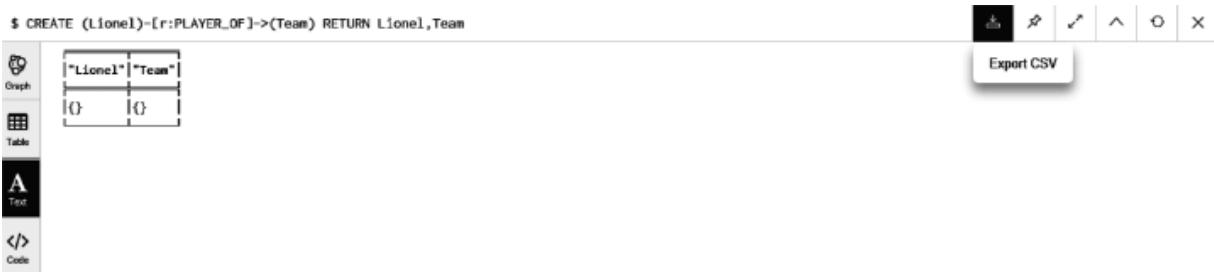
### *Output in text form:*

```
$ CREATE (Lionel)-[r:PLAYER_OF]->(Team) RETURN Lionel,Team
```



The screenshot shows the Neo4j browser interface. On the left, there is a vertical sidebar with four tabs: 'Graph' (selected), 'Table', 'Text', and 'Code'. The main area displays a relationship graph with two nodes: 'Lionel' and 'Team'. The 'Lionel' node has a label 'Lionel' and no properties. The 'Team' node has a label 'Team' and no properties. A relationship 'r:PLAYER\_OF' connects them. The 'Text' and 'Code' tabs are empty.

One can also download (export) the graph in the format you want to save it. Click on the download button, see the example (to export to csv):



The screenshot shows the Neo4j browser interface with the 'Graph' tab selected. The main area displays the same relationship graph as before. In the top right corner, there is a toolbar with several icons: a download icon, a refresh icon, a checkmark icon, a double arrow icon, a circular arrow icon, and a close icon. Below the toolbar, there is a button labeled 'Export CSV'.

### 5.3.1 Create a Relationship with Label and Properties

*Syntax:*

```
CREATE  (node1)-[label:Rel_Type {key1:value1,  key2:value2,  . . .  n}]->
(node2)
```

**Example:** Let us take an example to create a relationship for a node with label and properties using the CREATE statement.

First create a node “Seema” having multiple labels such as “person” and “author”.

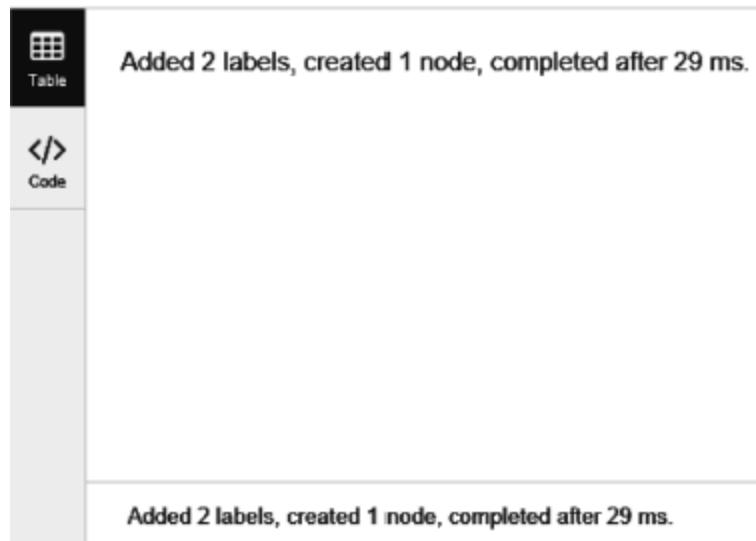
**Command:**

---

```
$ CREATE (Seema:person:author)
```

**Output:**

```
$ CREATE (Seema:person:author)
```



The screenshot shows the Neo4j Browser interface. On the left, there is a sidebar with two items: 'Table' and 'Code'. The 'Code' item is selected, indicated by a dark grey background. In the main content area, there are two horizontal lines, each containing the text 'Added 2 labels, created 1 node, completed after 29 ms.' This indicates that the command was executed successfully and a new node was created with two labels.

---

```
Added 2 labels, created 1 node, completed after 29 ms.
```

---

```
Added 2 labels, created 1 node, completed after 29 ms.
```

Then create few properties (name, FirstBook, and Publisher) for the same node “Seema”:

**Command:**

```
1 CREATE (Seema:author{name: "Seema Acharya", FirstBook: "Fundamentals of Business Analytics", Publisher: "Wiley India"})  
2 RETURN Seema
```

**Output:**

```
$ CREATE (Seema:author{name: "Seema Acharya", FirstBook: "Fundamentals of Business Analytics", Publisher: "Wiley India"})-
```

Displaying 1 nodes, 0 relationships.

A node “Seema” is created.

Next, create a node “Tech” with label “Technology” and properties (name: “Data and Analytics”).

***Command:***

```
$ CREATE (Tech:Technology {name: "Data and Analytics"})
```

***Output:***

```
$ CREATE (Tech:Technology {name: "Data and Analytics"})
```

Added 1 label, created 1 node, set 1 property, completed after 254 ms.

```
$
```

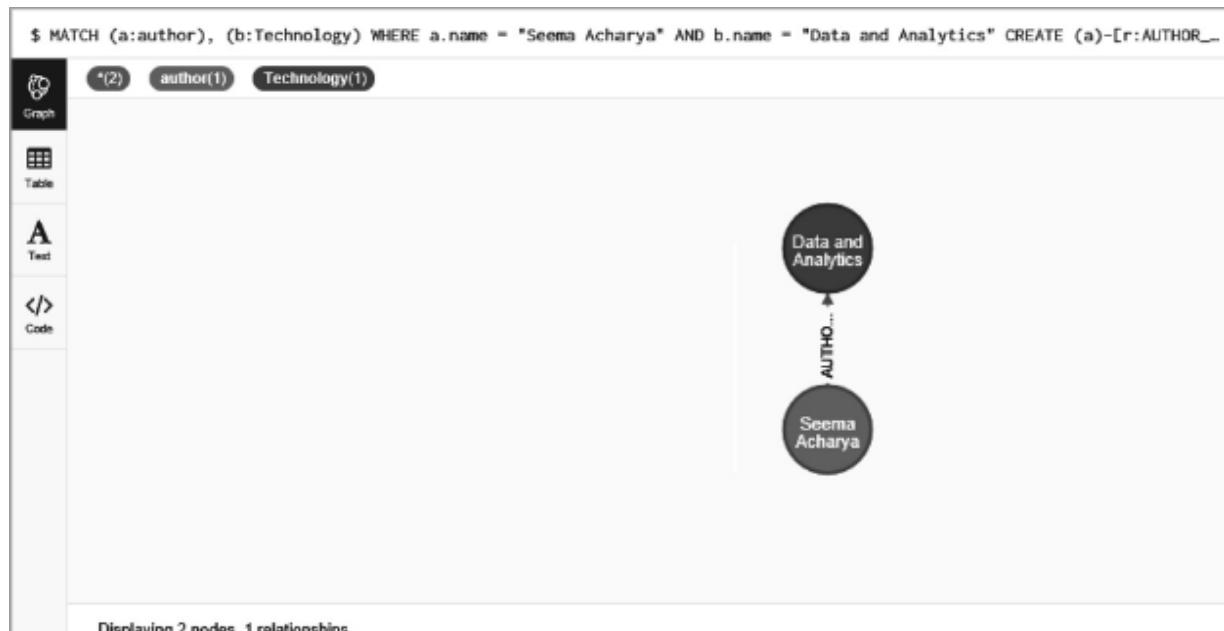
Added 1 label, created 1 node, set 1 property, completed after 254 ms.

Create a relationship “AUTHOR\_OF” between nodes “Seema” and “Tech” with labels “author” and “Technology”, respectively.

**Command:**

```
1 MATCH (a:author), (b:Technology) WHERE a.name = "Seema Acharya" AND b.name
  = "Data and Analytics"
2 CREATE (a)-[r:AUTHOR_OF {Books:5, VideoCourse:1}]->(b)
3 RETURN a,b
```

**Output:**



As can be seen from the screenshot above, a relationship “AUTHOR\_OF” is created between the two nodes “Seema” and “Tech”.

## 5.4 WHERE CLAUSE

The WHERE clause is used to specify a condition that helps to retrieve the exact data.

**Syntax:**

```
MATCH (label)
WHERE label.country = "property"
RETURN label
```

**Example:** To find the node with label “author” and property (FirstBook: “Fundamentals of Business Analytics”), use the following command.

**Command:**

```
1 MATCH (author)
2 WHERE author.FirstBook = "Fundamentals of Business Analytics"
3 RETURN author
```

**Output:**

\$ MATCH (author) WHERE author.FirstBook = "Fundamentals of Business Analytics" RETURN author

Graph

Table

Text

Code

\*(1) author(1)

author <id>: 119 FirstBook: Fundamentals of Business Analytics Publisher: Wiley India name: Seema Acharya

You can see it in tabular form to verify:

```
$ MATCH (author) WHERE author.FirstBook = "Fundamentals of Business Analytics" RETURN author
```

```
author
{
  "name": "Seema Acharya",
  "FirstBook": "Fundamentals of Business Analytics",
  "Publisher": "Wiley India"
}
Started streaming 1 records after 2 ms and completed after 2 ms.
```

### 5.4.1 WHERE Clause with Multiple Conditions

The “WHERE” clause can be used with multiple conditions.

*Syntax:*

```
MATCH (stu:Student)
WHERE stu.name = 'Abc' AND stu.class = '11 Grade'
RETURN stu
```

**Example:** Use the following query to filter the nodes in the Neo4j database using two conditions.

*Command:*

```
1 MATCH (author)
2 WHERE author.name = "Seema Acharya" AND author.Publisher = "Wiley India"
3 RETURN author
```

*Output:*

```
$ MATCH (author) WHERE author.name = "Seema Acharya" AND author.Publisher = "Wiley India" RETURN author
```

Graph

Table

Text

Code



author <id>: 119 **FirstBook**: Fundamentals of Business Analytics **Publisher**: Wiley India **name**: Seema Acharya

The above command using where clause with multiple conditions locates the node with properties (name: “Seema Acharya” and FirstBook: “Fundamentals of Business Analytics”).

### 5.4.2 COUNT() Function

In Neo4j database, the COUNT() function is used to count the number of rows.

*Syntax:*

```
1. MATCH (n {name: 'A'})-->(x)
2. RETURN n, count(*)
```

**Example:** Use the following query to demonstrate the usage of COUNT() function. It will count the number of books with “Wiley India” as publisher.

*Command:*

```
Match(n{Publisher: "Wiley India"})--(x
RETURN n, count(*)
```

*Output:*

```
$ Match(n{Publisher: "Wiley India"})--(x) RETURN n, count(*)
```

| n                                                                                                                           | count(*) |
|-----------------------------------------------------------------------------------------------------------------------------|----------|
| <pre>{   "name": "Seema Acharya",   "FirstBook": "Fundamentals of Business Analytics",   "Publisher": "Wiley India" }</pre> | 1        |

Started streaming 1 records after 4 ms and completed after 5 ms.

The output shows that there is only one book in the Neo4j database wherein the publisher is “Wiley India”. Therefore, count function returns 1.

## 5.5 CREATING A COMPLETE PATH

---

*Syntax:*

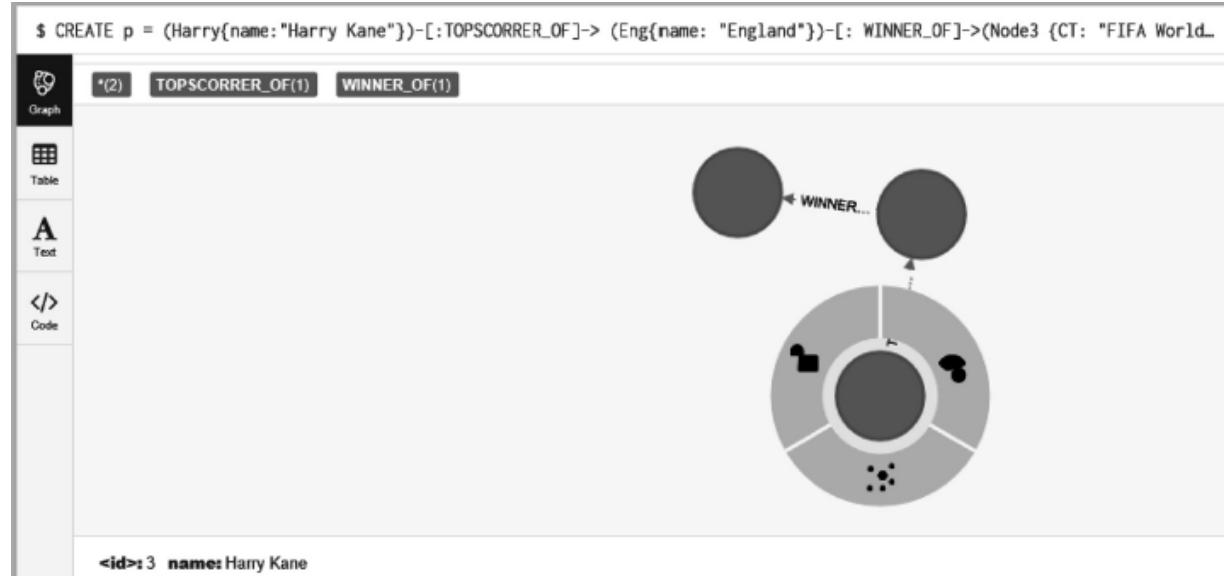
```
CREATE p = (Node1 {properties})-[:Relationship_Type]->
(Node2 {properties})[:Relationship_Type]->(Node3 {properties})
RETURN p
```

**Example:** In this example, we have three nodes – “Harry”, “Eng” and “Node3”. There are two relationships – “TOPSCORER\_OF” and “WINNER\_OF”.

*Command:*

```
1 CREATE p = (Harry{name: "Harry Kane"})-[:TOPSCORER_OF]->
2 (Eng{name: "England"})-[: WINNER_OF]->(Node3 {CT: "FIFA World Cup 2018"})
3 RETURN p
```

*Output:*



As can be seen, there are three nodes and two relationships “TOPSCORER\_OF” and “WINNER\_OF”.

## 5.6 CREATE INDEX

In Neo4j, as in other relational databases and NoSQL databases, an index is a data structure which is used to improve the speed of data retrieval operations.

*Where can an index be created?* An index can be created over a property on any node that has been given a label. Once an index is created, Neo4j will manage it and keep it up to date whenever the database is changed.

To create an index, use the CREATE INDEX ON statement.

**Example:**

```
$ CREATE INDEX ON :player(Goals)
```

**Output:**

```
$ CREATE INDEX ON :player(Goals)
```

```
Added 1 index, completed after 162 ms.
```

---

```
Added 1 index, completed after 162 ms.
```

**Verification:** The created schema and constraints become a part of database schema. Use the `:schema` command to check all indexes and constraints.

**Command:**

```
$ :schema
```

**Output:**

---

```
$ :schema
```

```
Indexes
ON :player(Goals) ONLINE
```

```
No constraints
```

## 5.7 CREATE CONSTRAINTS

---

In Neo4j, a constraint is used to place restrictions over the data that can be entered against a node or a relationship. There are two types of constraints in Neo4j:

1. **Uniqueness Constraint:** It specifies that the property must contain a unique value. For example, no two nodes with a player label can share a value for the Goals property.

**2. Property Existence Constraint:** It ensures that a property exists for all nodes with a specific label or for all relationships with a specific type.

In the example under consideration, “Seema” is the node, “author” is the label, and “FirstBook” is the property.

**Command:**

```
$ CREATE CONSTRAINT ON (Seema:author) ASSERT Seema.FirstBook IS UNIQUE
```

**Output:**

---

```
$ CREATE CONSTRAINT ON (Seema:author) ASSERT Seema.FirstBook IS UNIQUE
```



Table

Added 1 constraint, completed after 5257 ms.



Code

---

Added 1 constraint, completed after 5257 ms.

---

**Verification:** To verify if the constraint has been added successfully, use the :SCHEMA command.

---

```
$ :schema
```

```
Indexes
```

```
ON :player(Goals) ONLINE
ON :author(FirstBook) ONLINE  (for uniqueness constraint)
```

```
Constraints
```

```
ON ( author:author ) ASSERT author.FirstBook IS UNIQUE
```

**Note:** Property existence constraint is only available in Neo4j Enterprise edition.

## 5.8 SELECT DATA WITH MATCH

---

**Create a node:**

**Example:** Create a node “a” with label “Actors”, property (Name: “Leonardo Di Caprio”).

**Command:**

```
$ CREATE (a:Actors { Name : "Leonardo Di Caprio" })
```

**Output:**

---

|                                                                                              |                                                                                                                                   |
|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <br>Table | \$ CREATE (a:Actors { Name : "Leonardo Di Caprio" })<br><br>Added 1 label, created 1 node, set 1 property, completed after 38 ms. |
| <br>Code  | Added 1 label, created 1 node, set 1 property, completed after 38 ms.                                                             |

---

**Fetch single record:** To fetch the row/record on the basis of a property use the following code.

### ***Command:***

```
1 MATCH (a:Actors)
2 WHERE a.Name = "Leonardo Di Caprio"
3 RETURN a
```

### ***Output:***

---

\$ Match(a:Actors) Where a.Name = "Leonardo Di Caprio" Return a

|                                                                                          |                                                                                                  |
|------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
|  Graph  |  *(1) Actors(1) |
|  Table  |                                                                                                  |
|  Text   |                                                                                                  |
|  Code |                                                                                                  |



Leonardo Di Caprio

---

Displaying 1 nodes, 0 relationships.

---

## **5.9 FETCH ALL NODES**

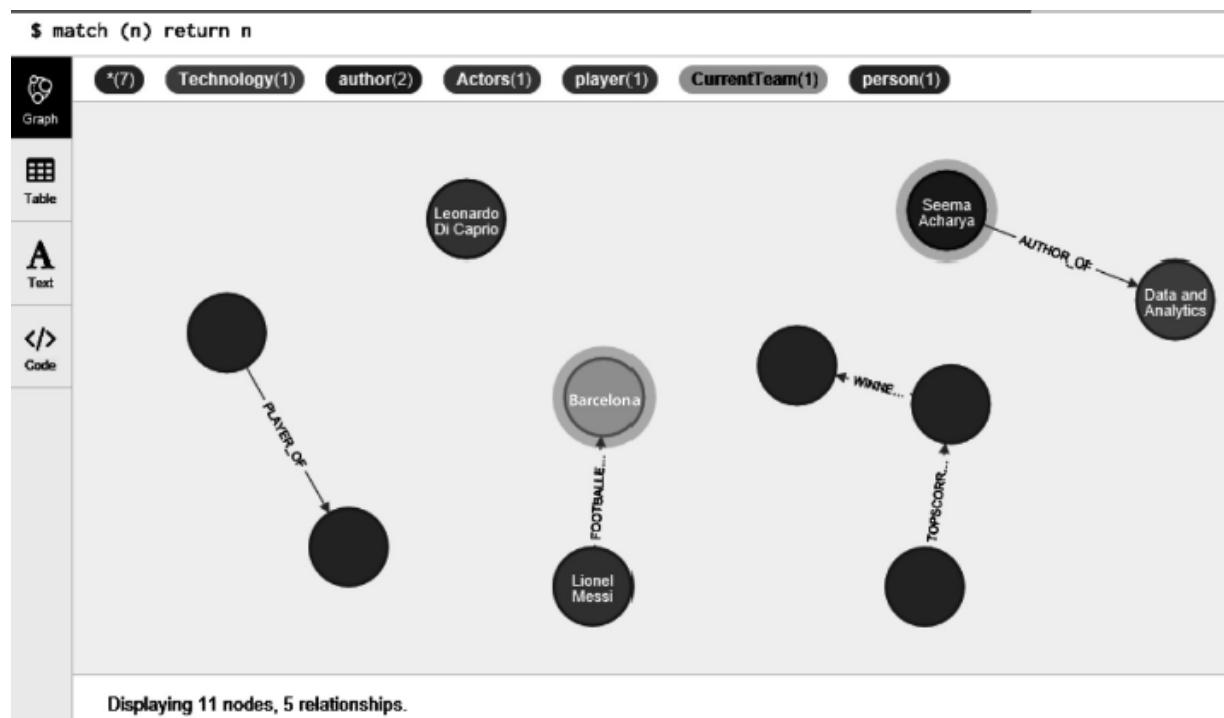
---

To retrieve all nodes from a database, just avoid the filters. Use the following code to retrieve all nodes from a database:

**Command:**

```
MATCH (n) RETURN n
```

**Output:**



You can also see it in tabular form.

```
$ match (n) return n
```

```
{  
  "name": "Seema Acharya",  
  "FirstBook": "Fundamentals of Business  
Analytics",  
  "Publisher": "Wiley India"  
}  
  
{  
  "Name": "Leonardo Di Caprio"  
}  
  
{  
  "name": "Lionel Messi",  
  "YOB": 1987,  
  "POB": "Argentina"  
}  
  
Started streaming 11 records in less than 1 ms and completed after 1 ms.
```

## 5.10 DROP AN INDEX

In Neo4j, “DROP INDEX ON” statement is used to drop an index from the database. It will permanently remove the index from the database.

**Example:**

```
$ DROP INDEX ON :Player(Goals)
```

*Output:*

```
$ DROP INDEX ON :player(Goals)
```

|                                                                                         |                                        |
|-----------------------------------------------------------------------------------------|----------------------------------------|
|  Table | Removed 1 index, completed after 1 ms. |
|  Code  |                                        |

```
Removed 1 index, completed after 1 ms.
```

Verify that the index is removed using the below command.

*Command:*

```
$ :schema
```

*Output:*

```
$ :schema
```

|             |                                                          |
|-------------|----------------------------------------------------------|
| Indexes     | ON :author(FirstBook) ONLINE (for uniqueness constraint) |
| Constraints | ON ( author:author ) ASSERT author.FirstBook IS UNIQUE   |

## 5.11 DROP A CONSTRAINT

DROP CONSTRAINT statement is used to drop or remove a constraint from the database as well as its associated index.

**Example:** Use the following statement to drop the previously created constraint and its associated index.

**Command:**

```
$ DROP CONSTRAINT ON (Seema:author) ASSERT Seema. FirstBook IS UNIQUE
```

**Output:**

```
$ DROP CONSTRAINT ON (Seema:author) ASSERT Seema. FirstBook IS UNIQUE
```



Table



Code

Removed 1 constraint, completed after 4 ms.

Removed 1 constraint, completed after 4 ms.

Verify that the constraint is dropped using the :schema command.

```
$ :schema
```

```
No indexes
```

```
No constraints
```

## 5.12 DELETE A NODE

---

In Neo4j, DELETE statement is always used with MATCH statement to delete whatever data is matched. The DELETE command is used at the same place where we used the RETURN clause in our previous examples.

**Example:**

```
$ MATCH (Leonardo:actor{Name: "Leonardo Di Caprio"}) DELETE Leonardo
```

*Output:*

---

```
$ MATCH (Leonardo:Actors{Name: "Leonardo Di Caprio"}) DELETE Leonardo
```

|                                                                                           |                                         |
|-------------------------------------------------------------------------------------------|-----------------------------------------|
|  Table | Deleted 1 node, completed after 557 ms. |
|  Code  | Deleted 1 node, completed after 557 ms. |

## 5.13 DELETE MULTIPLE NODES

---

You can delete multiple nodes by using MATCH and DELETE commands in a single statement. You just have to put the different nodes separated by a column.

Let us start with creating two nodes:

1. Node a with label “Student” and property (Name: “Christopher Grey”)

```
$ CREATE (a:Student {Name: 'Christopher Grey'})
```

2. Node b with label “Employee” and property (Name: “Mark Simpson”)

```
$ CREATE (b:Employee {Name: 'Mark Simpson'})
```

Now we delete both the nodes a, b with labels “Student” and “Employee”.

1. MATCH (a:Student {Name: ‘Christopher Grey’}), (b:Employee {Name: ‘Mark Simpson’})
2. DELETE a, b

```
$ MATCH (a:Student {Name: 'Christopher Grey'}), (b:Employee {Name: 'Mark Simpson'}) DELETE a,b
```

The screenshot shows a database interface with a sidebar on the left containing 'Table' and 'Code' buttons. The main area displays a message: 'Deleted 2 nodes, completed after 33 ms.' This message appears twice, once above and once below the interface's header. The 'Code' button is currently selected.

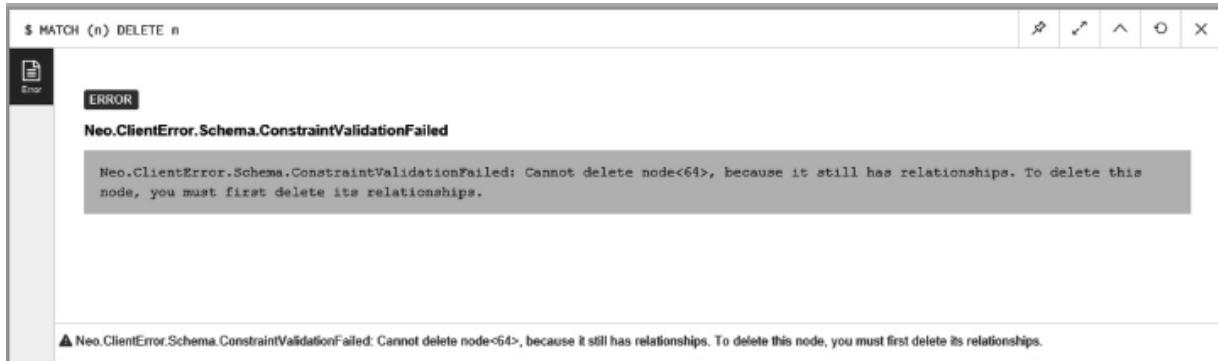
## 5.14 DELETE ALL NODES

---

To delete all nodes from the database do not use any filter criteria.

1. MATCH (n) **DELETE n**

**Note:** The above statement cannot delete nodes if they have any relationships. In other words, you must delete any relationships before you delete the node itself, otherwise you will get the following error message.



The screenshot shows a Neo4j browser interface. In the top-left, there is a text input field containing the query: '\$ MATCH (n) DELETE n'. To the right of the input field is a toolbar with icons for search, refresh, and other operations. Below the input field, a dark grey bar displays the word 'ERROR' in white. Underneath this bar, the error message 'Neo.ClientError.Schema.ConstraintValidationFailed' is shown in a white box. A detailed error message follows: 'Neo.ClientError.Schema.ConstraintValidationFailed: Cannot delete node<64>, because it still has relationships. To delete this node, you must first delete its relationships.' At the bottom of the browser window, there is a status bar with the same error message: 'Neo.ClientError.Schema.ConstraintValidationFailed: Cannot delete node<64>, because it still has relationships. To delete this node, you must first delete its relationships.'

There is a method to delete a node and all relationships related to that node. Use the DETACH DELETE statement:

**Command:**

```
$ MATCH (Seema:author{name: "Seema Acharya"}) DETACH DELETE Seema
```

**Output:**

```
$ MATCH (Seema:author{name: "Seema Acharya"}) DETACH DELETE Seema
```



Table

Deleted 1 node, deleted 1 relationship, completed after 378 ms.



Code

Deleted 1 node, deleted 1 relationship, completed after 378 ms.

## 5.15 DELETE A RELATIONSHIP

Deleting relationship is as simple as deleting nodes. Use the MATCH statement to match the relationships you want to delete.

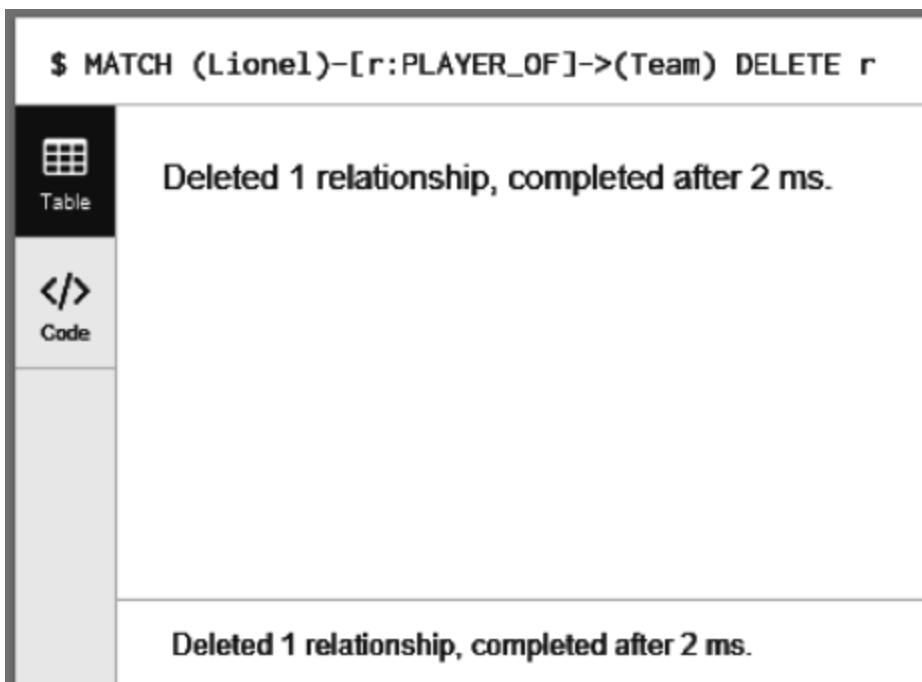
You can delete one or many relationships or all relationships by using one statement.

**Example:** Delete the relationship named “PLAYER\_OF” from the database:

**Command:**

```
1. MATCH (Lionel)-[r:PLAYER_OF]->(Team)  
2. DELETE r
```

**Output:**



The screenshot shows the Neo4j browser interface. On the left, there is a sidebar with two buttons: 'Table' (selected) and 'Code'. The main area displays the result of a Cypher query: '\$ MATCH (Lionel)-[r:PLAYER\_OF]->(Team) DELETE r'. Below this, the message 'Deleted 1 relationship, completed after 2 ms.' is shown twice, once for each row of the table.

| Deleted 1 relationship, completed after 2 ms. |
|-----------------------------------------------|
| Deleted 1 relationship, completed after 2 ms. |

To delete all relationships and to delete all nodes, use the following two statements. The first statement will delete all relationships and the second one will delete all nodes.

***Commands:***

```
MATCH ()-[r]-() DELETE r
MATCH (n) DELETE n
```

## 5.16 MERGE COMMAND

Neo4j MERGE command is a combination of CREATE and MATCH commands. This command is used to search for a given pattern in the graph. If it exists in the graph then it will return the result, otherwise it creates a new node/relationship and returns the results.

Using the MERGE command you can do the following things:

1. Merge a node with label.
2. Merge a node with properties.
3. Merge a relationship.

***Syntax:***

```
MERGE (node:label {properties . . . . .})
```

### Example:

To get started, let us create two nodes and a relationship:

1. Node “FOBA” with label “book” and properties (name: “Fundamentals of Business Analytics”, YOP: 2011).
2. Node “Pub” with label “Publisher” and property (name: “Wiley India”).
3. Create a relationship “Book\_From” between “FOBA” and “Pub”.

### Commands:

```
1. CREATE (FOBA:book{name: "Fundamentals of Business Analytics", YOP: 2011})  
2. CREATE (Pub:Publisher {name: "Wiley India"})  
3. CREATE (FOBA)-[r:Book_From]->(Pub)
```

### Output:

```
$ CREATE (FOBA:book{name: "Fundamentals of Business Analytics", YOP: 2011}) CREATE (Pub:Publisher {name: "Wiley India"}) CREATE (FOBA)-[r:Book_From]->(Pub)  
Added 2 labels, created 2 nodes, set 3 properties, created 1 relationship, completed after 62 ms.
```

---

```
Added 2 labels, created 2 nodes, set 3 properties, created 1 relationship, completed after 62 ms.
```

### Merge a Node with a Label

MERGE clause is used to merge a node in the database based on the label. When you try to merge a node based on the label, then Neo4j verifies if there exists any node with the given label. If the node with the specified label does not exist, then the current node will be created.

### Syntax:

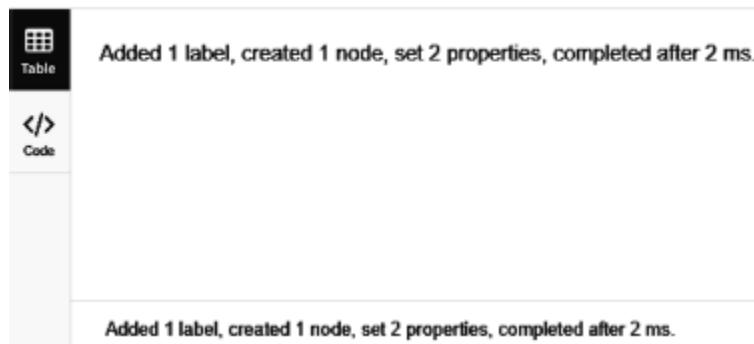
```
1. MERGE (node:label) RETURN node
```

### Example:

Let us merge a node “BDA” to a label Publisher. Neo4j verifies if there is any node with the label Publisher. If not, it creates a node named “BDA” and returns it.

If any node with the given label exists, Neo4j returns them all.

```
$ CREATE (BDA:book{name: "Big Data and Analytics", YOP: 2015})
```

A screenshot of the Neo4j browser interface. On the left, there is a sidebar with a 'Table' icon and a 'Code' icon. The main area shows a message: 'Added 1 label, created 1 node, set 2 properties, completed after 2 ms.' This message is repeated below it. The background of the main area is light gray.

A node “BDA” is created with label “book”.

Execute the below statement:

```
$ MERGE (BDA:Publisher) RETURN BDA
```

***Output:***

```
$ MERGE (BDA:Publisher) RETURN BDA
```

Graph \*(1) Publisher(1)

Table

Text

Code

Wiley India

Displaying 1 nodes, 0 relationships.

The node “BDA” is merged with label “Publisher”.

### TRY THIS – 1

1. It is open source, schema free, NoSQL and a popular graph database. What is “it”?
  - (a) Redis
  - (b) MongoDB
  - (c) Cassandra
  - (d) Neo4j
2. Which query language is used by Neo4j?
  - (a) SQL
  - (b) Cassandra query language
  - (c) Cipher query language
  - (d) All of the above

3. In which scenario is Neo4j widely used?
- (a) Highly connected data social network
  - (b) Recommendation (e.g., e-commerce)
  - (c) Path Finding
  - (d) All of the above
- 

## REMEMBER ME



1. With graph database, it is easier to represent relationships between nodes.
2. Few typical use cases of graph databases are:
  - (a) Access control lists
  - (b) Recommendation engines
  - (c) Social networks
  - (d) Interconnected data
3. Few examples of graph databases are:
  - (a) Neo4j
  - (b) Allegrograph
  - (c) OrientDB
  - (d) InfiniteGraph
4. Syntax to check for nodes:

```
MATCH (n) RETURN n;
```

5. To create multiple nodes, the syntax of the command is

```
CREATE (node1), (node2), (node1),  
(node n) ..
```

6. To create a node with a label, the syntax is

```
CREATE (node:label)
```

7. Syntax to create a node with properties:

```
CREATE (node:label { key1: value,  
key2: value, . . . . . })
```

8. To create a relationship between nodes, the syntax is

```
CREATE (node1)-[:RelationshipType]
->(node2)
```

## 9. Syntax to Create a Relationship with Label and Properties

```
CREATE (node1)-[label:Rel_Type
{key1:value1, key2:value2, . . n}]
-> (node2)
```

## 10. The WHERE clause is used to specify a condition that helps to retrieve the exact data.

### *Syntax:*

```
MATCH (label)
WHERE label.country = "property"
RETURN label
```

11. In Neo4j, as in other relational databases and NoSQL databases, an index is a data structure which is used to improve the speed of data retrieval operations.
12. In Neo4j, a constraint is used to place restrictions over the data that can be entered against a node or a relationship.

## QUESTION ME

---

1. List a few popular graph databases.
2. Why Neo4j is called a graph database?
3. Is it easy to fragment a Neo4j graph across multiple servers?
4. Is it possible to query Neo4j over the internet?
5. Explain the areas where Neo4j can be used?
6. Explain the differences between RDBMS and graph database.
7. What is the role of building blocks like Nodes, Relationships, Properties and Labels in Neo4j?
8. What is MATCH command? Where is it used in Neo4j?
9. Explain some features of Neo4j.
10. Give the syntax for deleting all nodes and relationships in Neo4j.

## PRACTICE ME

---

1. Create the nodes and relationship to state that “Tiramisu” is an “Italian dessert”.

## Solution

Create the nodes with labels “Dessert” and “Country\_Dessert” and a relationship “Is-An” between them.

Let us look at the syntax for getting the desired output.

```
create(Node1:Dessert{name:  
"Tiramisu"})-[r:Is_An]->(Node2:  
Country_Dessert{name:"Italian_  
Dessert"}) ;
```

The above syntax creates two nodes:

- (a) Node1
- (b) Node2

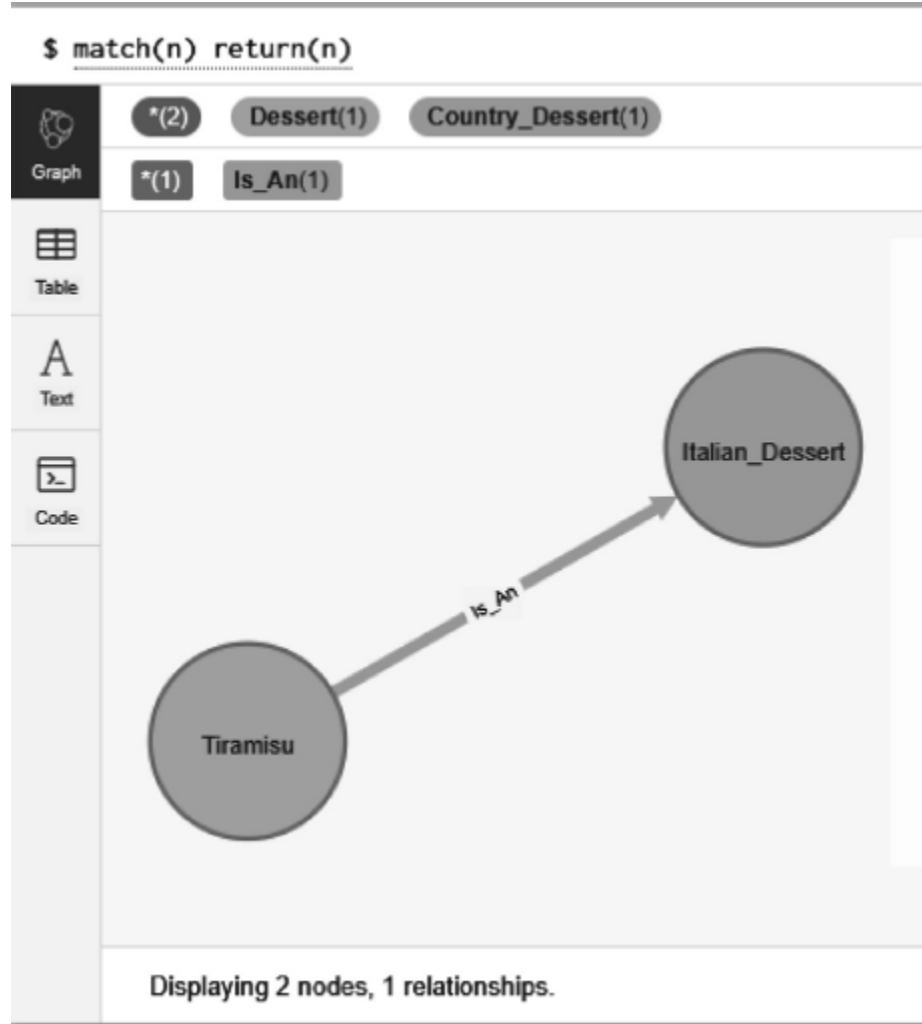
There are two labels:

- (a) “Dessert” label for Node1
- (b) Country\_Dessert label for Node2

“Node1” has property “name: Tiramisu”. Node2 has property, “name: Italian\_Dessert”. “Is\_An” is a relationship between nodes “Node1” and “Node2”.

```
$ create(Node1:Dessert{name:"Tiramisu"})-[r:Is_An]->(Node2:Country_Dessert{name:"Italian_Dessert"});
```

|                                                                                           |                                                                                                   |
|-------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
|  Table | Added 2 labels, created 2 nodes, set 2 properties, created 1 relationship, completed after 14 ms. |
|  Code  | Added 2 labels, created 2 nodes, set 2 properties, created 1 relationship, completed after 14 ms. |



2. Depict using Neo4j cipher language, an Employee with EmployeeID="E1001" who works in three projects, namely, "P1001", "P1002", "P1003".

### Solution

Run the below commands in a batch.

```
Create (Emp:Employee{EmployeeID:  
"E1001"})  
Create (Proj1:Project{ProjectID:  
"P1001"}), (Proj2:Project  
{ProjectID: "P1002"}),  
(Proj3:Project{ProjectID:  
"P1003"})
```

```
Create (Emp)-[r1:works_in]->(Proj1)  
Create (Emp)-[r2:works_in]->(Proj2)  
Create (Emp)-[r3:works_in]->(Proj3)
```

```
$ Create (Emp:Employee{EmployeeID:"E1001"}) Create(Proj1:Project{ProjectID:"P1001"}), (Proj2:Project{ProjectID:"P1002"}), (Proj3:Project{ProjectID:"P1003"}), (Proj4:Project{ProjectID:"P1004"})
```



```
$ match (n) return (n);
```



3. A csv file “artists.csv” has data about four pop/rock bands and the year that the band was formed. Read the data into Neo4j and return the rows (data) contained therein.

### Solution

```
LOAD CSV FROM 'https://neo4j.com/docs/cypher-manual/3.5/csv/artists.csv' AS row
RETURN row
```

```
$ LOAD CSV FROM 'https://neo4j.com/docs/cypher-manual/3.5/csv/artists.csv' AS row RETURN row
```

| Table | row                              |
|-------|----------------------------------|
| Text  | [{"1", "ABBA", "1992"}]          |
| Text  | [{"2", "Roxette", "1986"}]       |
| Text  | [{"3", "Europe", "1979"}]        |
| Text  | [{"4", "The Cardigans", "1992"}] |

Started streaming 4 records after 4 ms and completed after 289 ms.

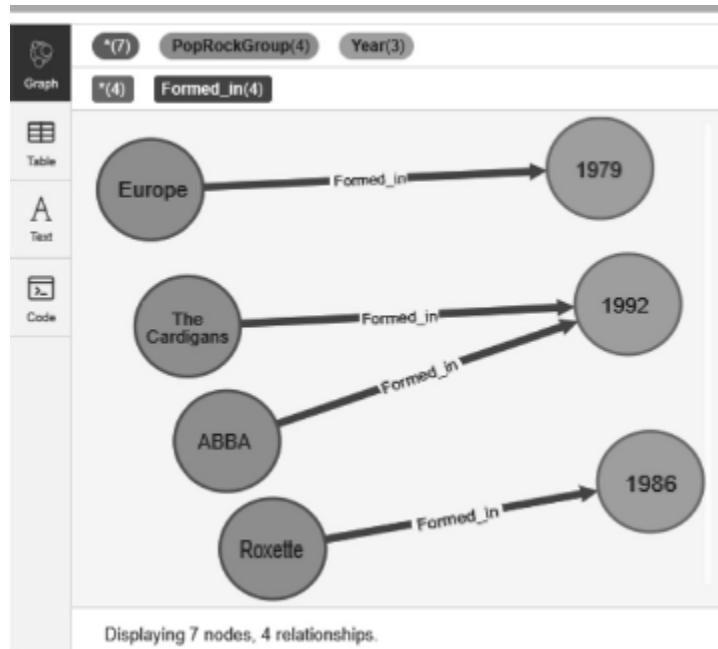
4. A csv file “artists.csv” has data about four pop/rock bands and the year that the band was formed. Read the data into Neo4j. Create nodes for bands and the years in which they were formed. Link the band with the year of their formation using relationship “Formed\_in”.

### Solution

A csv file “artists.csv” has data about four pop/rock bands and the year that the band was formed. Read the data into Neo4j. Create nodes for bands and the years in which they were formed. Link the band with the year of their formation using relationship “Formed\_in”.

The syntax to achieve the same is as given.

```
LOAD CSV FROM 'https://neo4j.com/
docs/cypher-manual/3.5/csv/art-
ists.csv' AS row
MERGE (sub:PopRockGroup
{name:row[1]})
MERGE (sup:Year {name:row[2]})
CREATE (sub)-[r:Formed_in]->
(sup)
return sub, sup, row
```



5. What is the interpretation of the below query?

```

MATCH (p:Person { name:"Homer
Flinstone" })
RETURN p

```

After analyzing the above statements, answer the questions:

- (a) What is the label for the node?
- (b) What is/are the property for the node?
- (c) What will the statement/command return?

### Solution

- (a) The label for the node is “Person”.
- (b) The node has only one property, “name:” “Homer Flinstone”.
- (c) The statement will return the node, “p”.

6. When will the below output be returned?

```
$ MATCH (p:Person { name:"Homer Flintstone" }) RETURN p
```

(no changes, no records)

Completed after 1 ms.

## Solution

This output will be returned when there is no such node “p” with a label “Person” and property name: “Homer Flintstone”.

7. What will the below command do?

```
CREATE (b:Album { Name : "Heavy  
as a Really Heavy Thing",  
Released : "1995" })  
RETURN b
```

## Solution

It will create a node “b” with label “Album” and two properties: (a) Name: “Heavy as a Really Heavy Thing” and (b) Released: “1995”.

```
$ CREATE (b:Album { Name : "Heavy as a Really Heavy Thing", Released : "1995" }) RETURN b
```

Graph

Table

Text

Code

Album(1)

Heavy as a  
Really Heavy  
Thing

Displaying 1 nodes, 0 relationships.

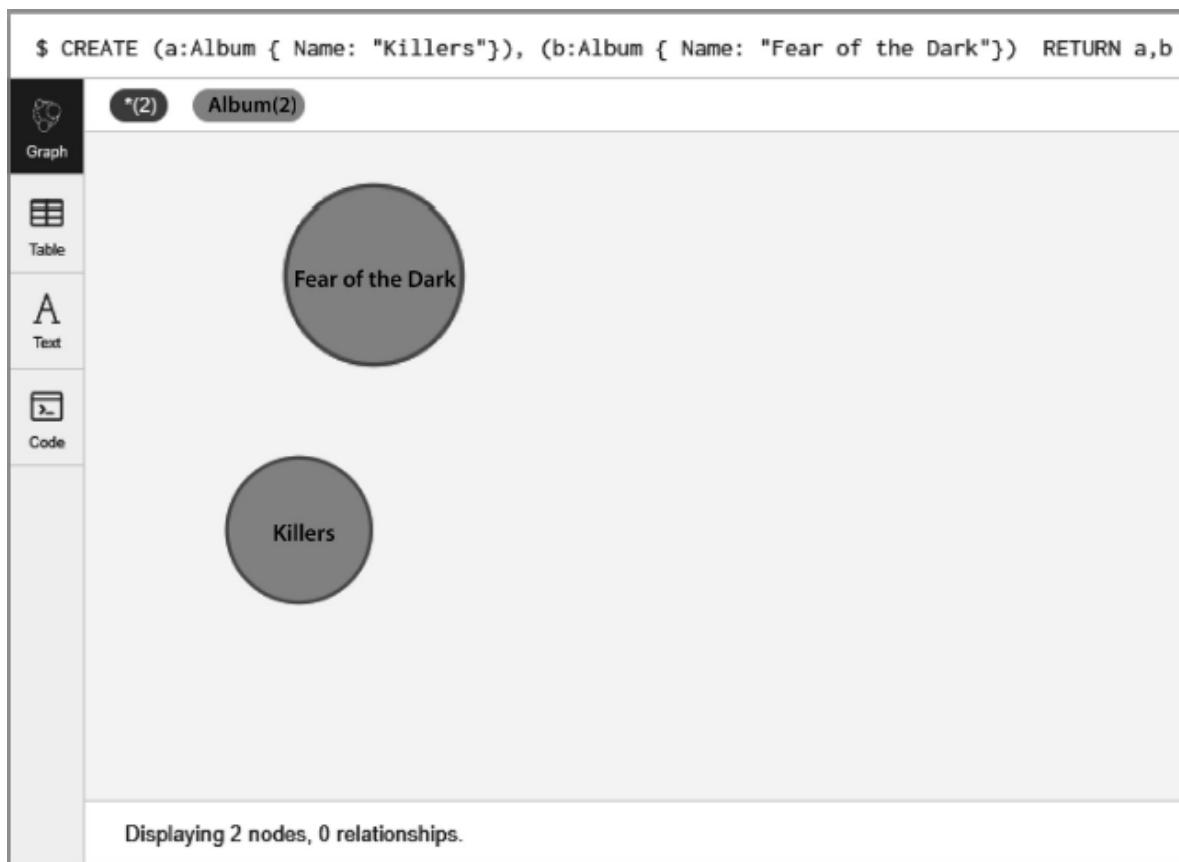
8. Is there any difference between the following two sets of commands in terms of the output that they will produce?

(a) `CREATE (a:Album { Name: "Killers")}, (b:Album { Name: "Fear of the Dark"})  
RETURN a,b`

(b) `CREATE (a:Album { Name: "Killers")  
CREATE (b:Album { Name: "Fear of the Dark"})  
RETURN a,b`

### Solution

Both will return the same output.



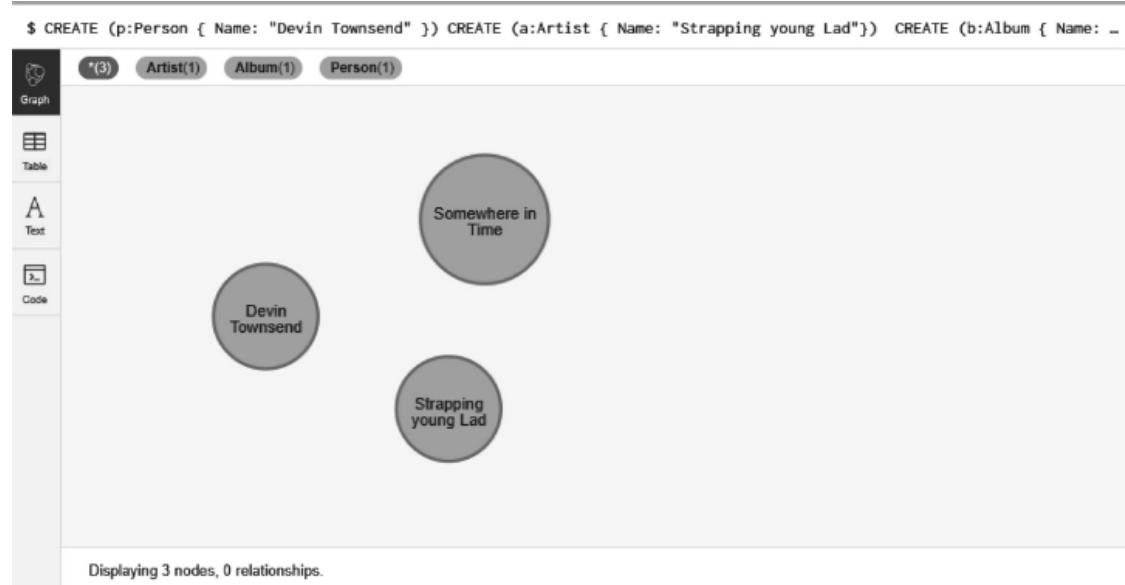
9. What will be the output of the below statements executed in sequence?

(a) CREATE (p:Person { Name: "Devin Townsend" })  
 CREATE (a:Artist { Name: "Strapping young Lad" })  
 CREATE (b:Album { Name: "Somewhere in Time" })  
 RETURN a,b, p

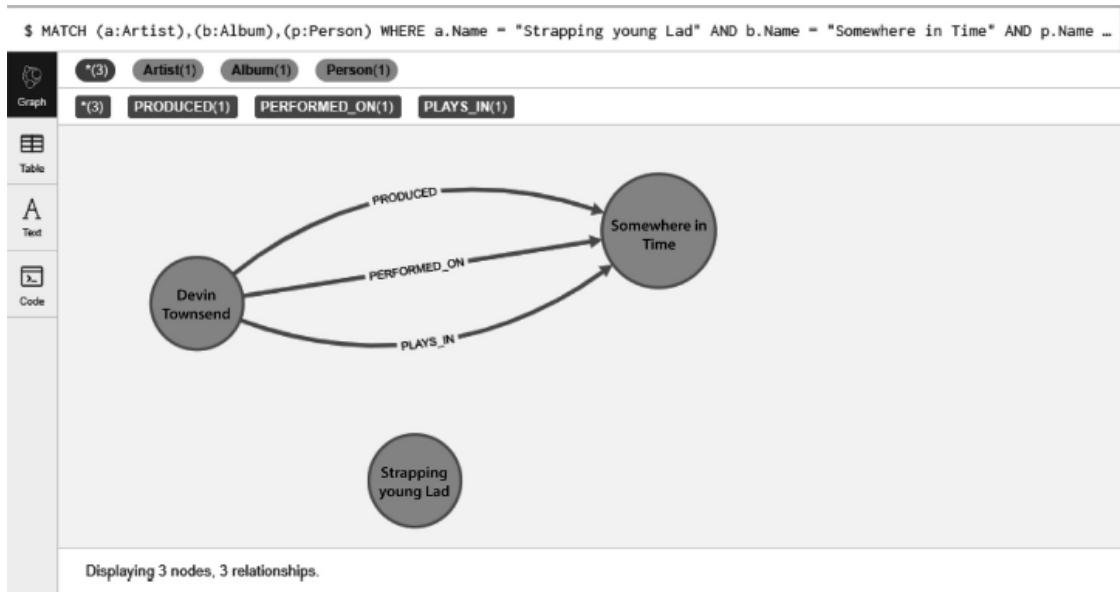
(b) MATCH (a:Artist),  
 (b:Album), (p:Person)  
 WHERE a.Name = "Strapping young  
 Lad" AND b.Name = "Somewhere  
 in Time" AND p.Name = "Devin  
 Townsend"  
 CREATE (p)-[pr:  
 PRODUCED]->(b), (p)-[pf:  
 PERFORMED\_ON]->(b),  
 (p)-[pl:PLAYS\_IN]->(b)  
 return a,b,p,pr,pf,pl

## Solution

(a)



(b)



10. Create an index on the “Name” property of all the node with label “Album”.

### Solution

The syntax to be used is

```
create index on :Album(Name);
```

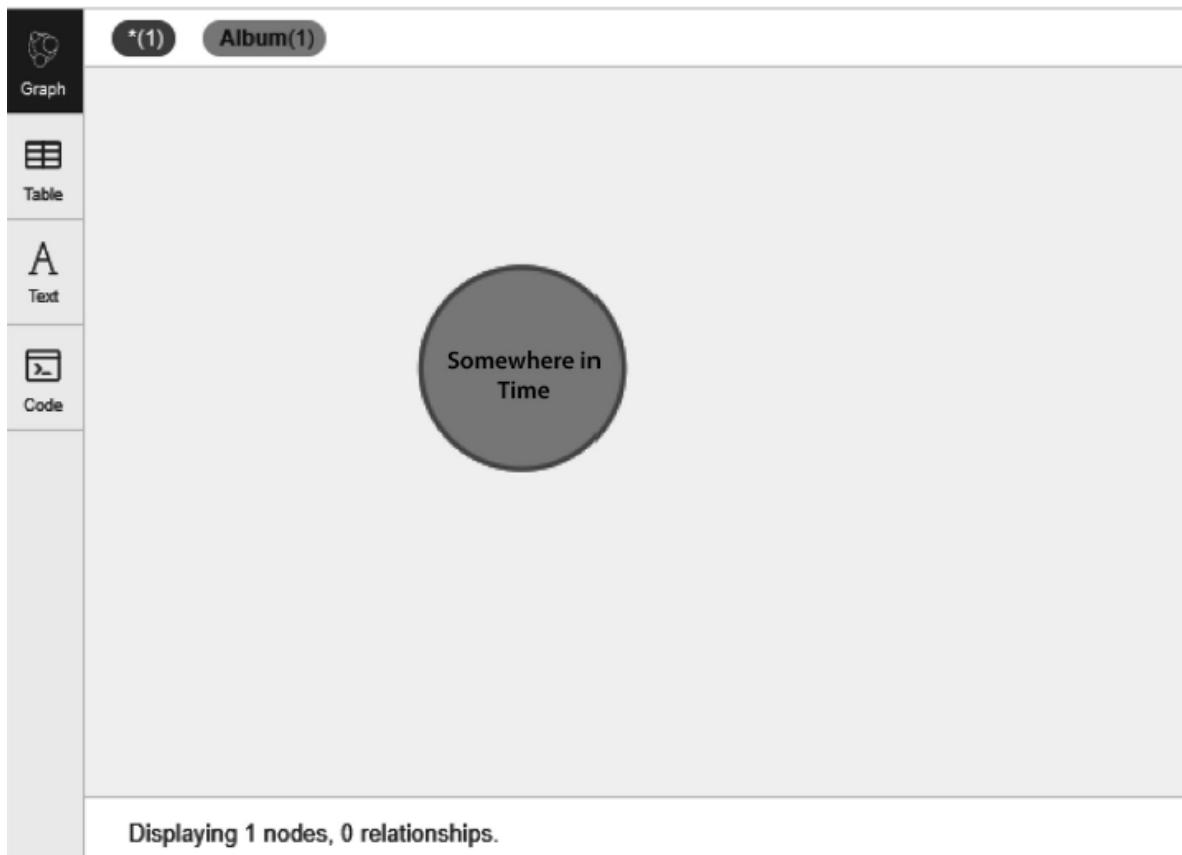
11. What is your understanding of the following command? What are we trying to achieve?

```
Match (a:Album {Name:
"Somewhere in Time"})
Using Index a:Album(Name)
Return a
```

### Solution

It will fetch the node with label “Album” and property as (Name: “Somewhere in Time”)

```
$ Match (a:Album {Name: "Somewhere in Time"}) Using Index a:Album(Name) Return a
```

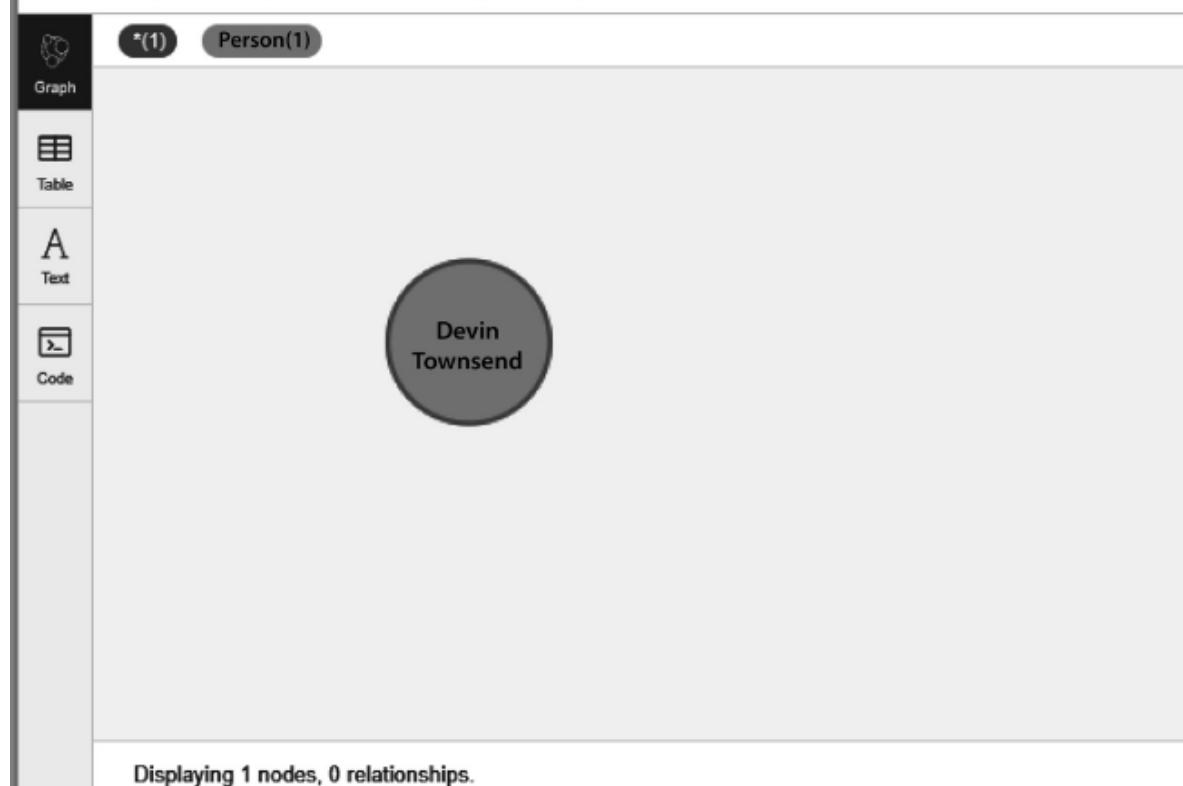


12. What will be the output of the following command?

```
Match (a:Person) -  
[:PRODUCED]->(b:Album)  
Where b.Name = "Somewhere in  
Time"  
Return a
```

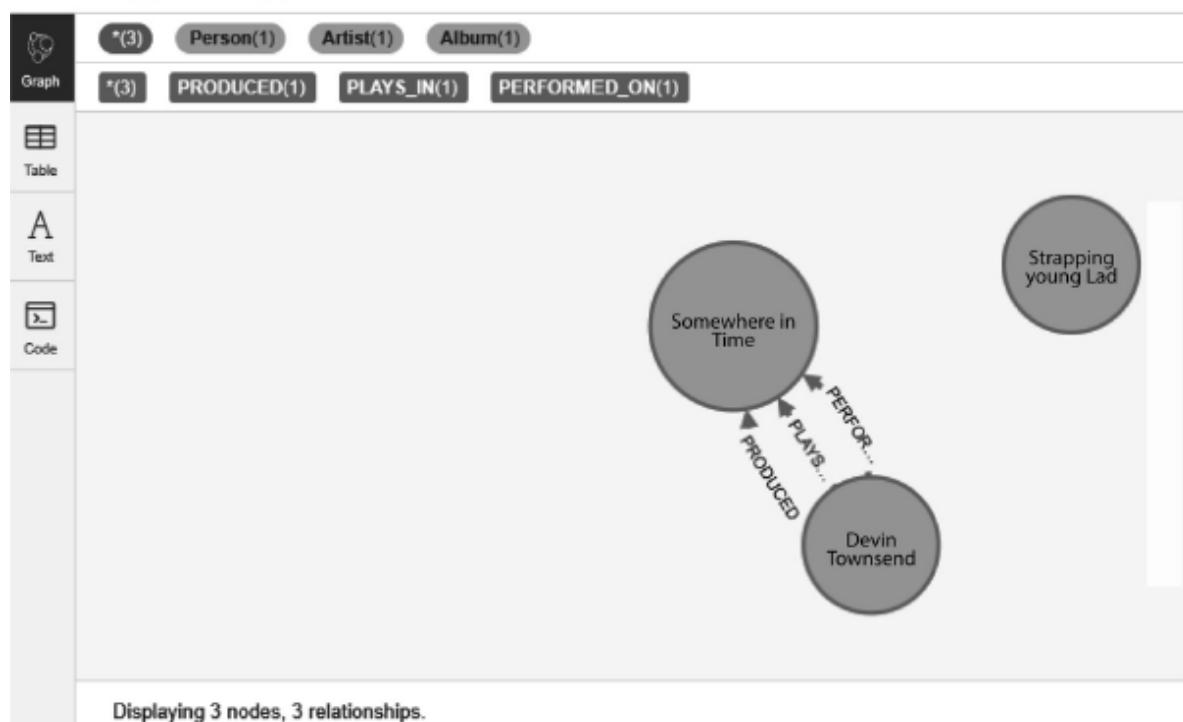
**Solution**

```
$ Match (a:Person) -[:PRODUCED]->(b:Album)  Where b.Name = "Somewhere in Time" Return a
```



13. Assume we have the following:

```
$ match (n) return (n)
```

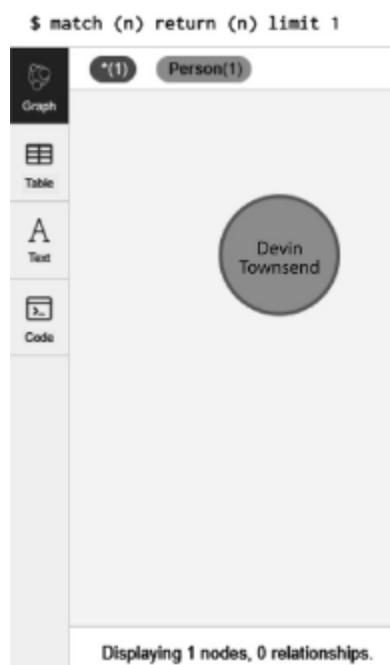


What will be the output of the following?

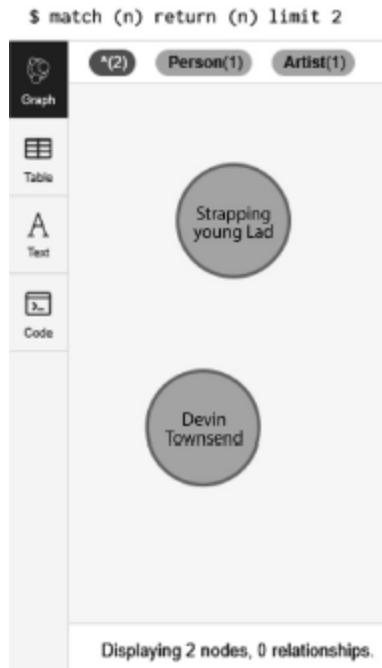
- (a) match (n) return (n) limit 1
- (b) match (n) return (n) limit 2
- (c) match (n) return (n) limit 3

### Solution

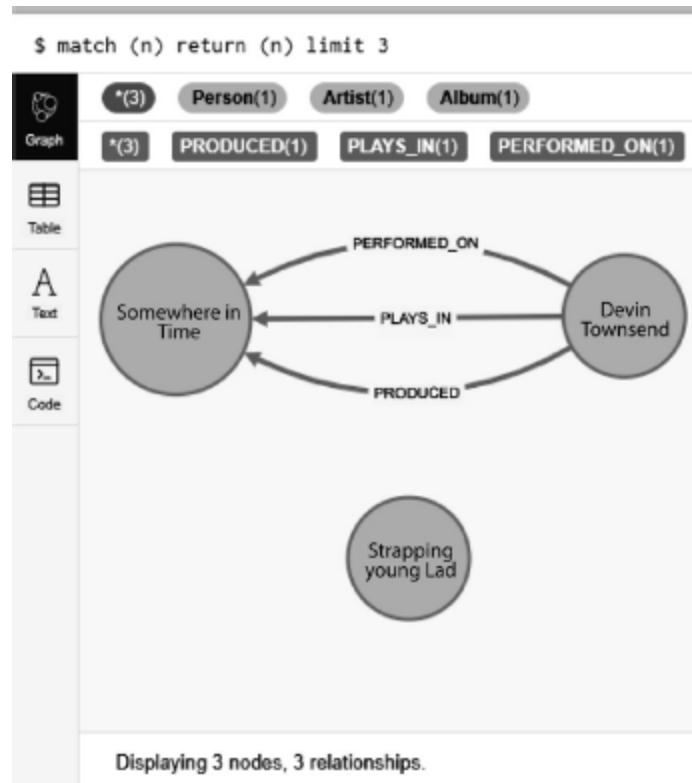
(a)



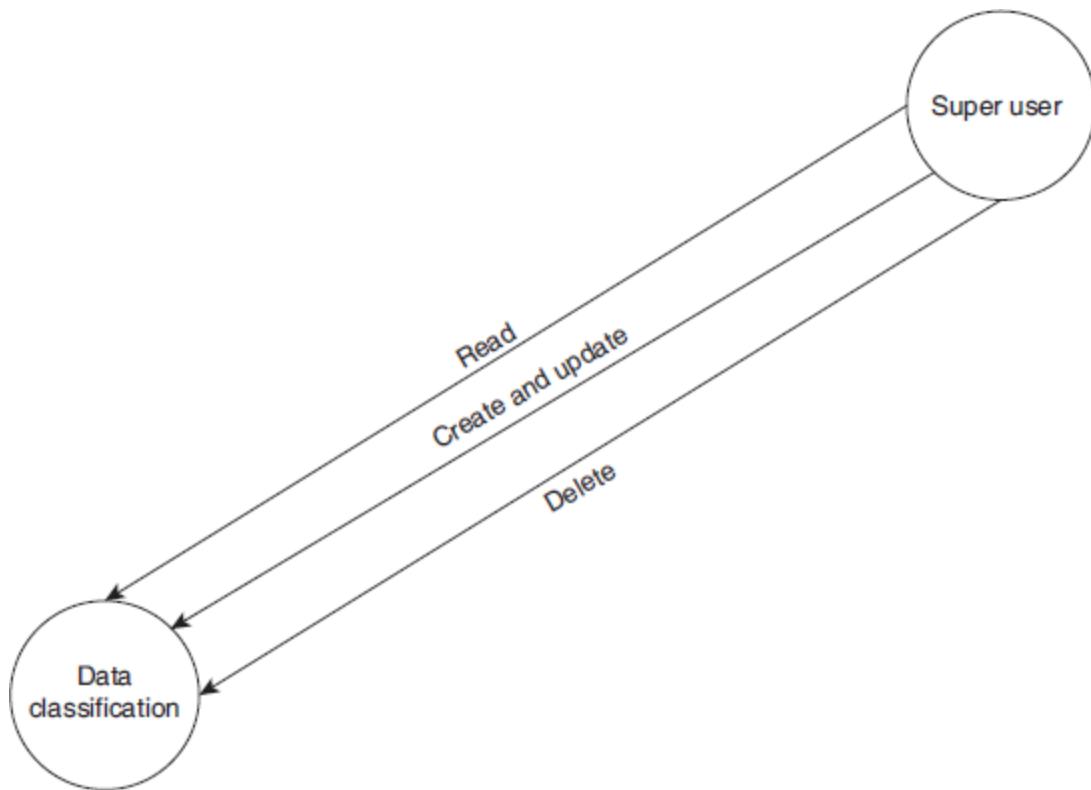
(b)



(c)



14. Create the following graph using Neo4j Cipher.



## REFERENCE ME

---

1. <https://neo4j.com/>
2. <https://en.wikipedia.org/wiki/Neo4j>
3. <https://github.com/neo4j/neo4j>
4. <https://neo4j.com/download/>

## ANSWERS

---

### Try This – 1

1. (d) Neo4j.

**Reason:** Redis is a key-value store, MongoDB is document database and Cassandra is a column family store.

2. (c) Cipher query language

**Reason:** SQL is used in RDBMS and Cassandra query language is used by Cassandra.

**3.** (d) All of the above

**Reason:** Neo4j is used to depict relationships between data.

*OceanofPDF.com*



# NoSQL Database Orientation

---

## BRIEF CONTENTS

- RDBMS or NoSQL?
  - NoSQL versus SQL
- Key–Value Store
  - Pros
  - Limitations
  - Use Cases
  - Few Examples of Key–Value NoSQL Databases
- Column Family Store
- Document Store
- Graph Store
- Examples of NoSQL Databases
  - Key–Value Stores
  - Document Stores
  - Column Stores
  - Graph Stores

## 6.1 RDBMS OR NoSQL?

---

One of the biggest decisions an organization is required to make today when it comes to choosing the right database is whether to invest in relational database (SQL) or a NoSQL database. Both SQL and NoSQL have their own clear advantages and disadvantages. This section highlights the major differences between SQL and NoSQL as well as the benefits of each.

**1. *Schema:*** RDBMS (SQL) works with pre-defined schema. Even before you start working with data, it is mandatory to define the structure of the data. Pre-defined schema makes it highly restrictive as all data must follow the same structure. There is significant upfront preparation required on the data before one can actually start working with the data.

NoSQL databases however support dynamic schema and the data can be stored in many different ways: key-value pairs, document store, column store or organized as graph-based database. This allows extreme flexibility to work with data without having to first plan and define structure. Fields can be added as and when required. Also NoSQL efficiently caters to exploding volume, growing data variety, and increasing speed.

**2. *Scalability:*** Another big difference between SQL and NoSQL is their scalability. SQL databases are vertically scalable, which implies that the load on a single server can be increased by increasing components like RAM, SSD, or CPU. In contrast, NoSQL databases are horizontally scalable, which means that they can handle increased traffic simply by adding more servers to the database. NoSQL databases have the ability to become larger and much more powerful, making them the preferred choice for large or constantly evolving datasets.

**3. *Community Support:*** RDBMS boasts of a very good and strong community support as it has been in existence from 1970s. However, NoSQL is relatively new and evolving, therefore the community support is not as well defined.

NoSQL databases have numerous benefits, including lower cost, open-source availability, and easier scalability, which makes NoSQL an appealing option for anyone thinking about integrating in Big Data. We have several choices when it comes to NoSQL. Should you be using key-value, document database, or column-oriented database or is graph-based database a better choice.

Let us take the example of e-commerce to understand the selection of various types of NoSQL databases basis the business requirements of an e-commerce retailer.

Think Amazon!!!

As a visitor or a customer on an e-commerce website, what all do you do/can do?

Few of the activities that visitors/customers perform on the e-commerce site are:

1. Create or modify your profile (advertisement preferences, payment preferences, etc.).
2. Product navigation: This is by far the most important feature of an online e-commerce shop. Users must have the ability to browse and check different categories of products quickly and efficiently through a user-friendly navigation system.
3. Search for a particular product (specifying brand, color, size, features, price range, etc.). Having a clear and well-positioned search box enhances the quality of user's experience.
4. Place the product in the shopping cart. The user should be able to easily move a selected product into the shopping cart. Also if he/she changes his/her mind at a later point in time, it should be equally easy to remove the product from the shopping cart or save it for later.
5. Checkout – apply coupons. Once the user goes for checkout, he/she should be able to see all coupons that can be applied to his/her bill and it should be convenient to apply the selected coupons.
6. Add money to your wallet. Gone are the days when the only way to pay for your shopped products was via cash on demand (COD) or debit/credit cards. Today you can pay through net banking, wallet, gift vouchers, phone and mobile payments, money orders, etc.
7. Apply for order cancellation/refund: Just as the ease with shopping for items/products and moving it in/out of the shopping cart, users should also effortlessly be able to cancel an order either in its entirety or partially. They should be assisted to apply for replacement/refund. Even with refund, there should be more than one way to get the required refund.

What does all this mean for the database?

1. Millions of customers access the e-commerce site at any point in time. This means that the database should have the capability to handle multiple read/write requests in real time or near real time.
2. There is data about customers, products, sellers, vendors, etc. The database should be able to handle large volumes of data. The database should facilitate easy and safe storage and retrieval.
3. Deal with unstructured data – product images, text description, video demo of product, customer feedback, etc.
4. Data manipulation by customers/vendors. Customers may update their profile information, addresses, advertisement preferences, payment preferences, etc.

We need a database that can help us with all of the above and more, with *low latency* (latency can be described as the time interval between the stimulation and the response or in layman's terms as the time interval between the submission of a query/request and the response), *high throughput* (in database, throughput refers to the number of transactions per second), *personalization as per customer* (recommendation basis past shopping history, preferences, etc.)

The question here is: *Can our relational databases handle all of the above requirements?* Let us look at the challenges faced by relational databases and the possible solution using NoSQL databases.

| Requirement      | Challenges using SQL Databases                                                                                                 | Solution using NoSQL Databases                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Architecture     | Centralized – single node dependency                                                                                           | Distributed – set up multiple machines                                                          |
| Massive storage  | Experience challenges with handling large volumes (terabytes, petabytes and exabytes of data which are now becoming the norm). | Can easily store terabytes, petabytes, exabytes of data using low-cost commodity hardware.      |
| High reliability | Requests cannot be processed if the central server is down.                                                                    | Data is copied to multiple nodes (replication) overcoming the problem with single node failure. |

| Requirement  | Challenges using SQL Databases                                                                                                                                                                                                                                             | Solution using NoSQL Databases                                                                                                                                                                                                                                                                                                                       |
|--------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data model   | <p>Fixed/rigid schema. Schema needs to be pre-defined before shoving the data in.</p> <p>Often times, tables do not map to objects in applications very well.</p> <p>Also they are not good at modelling certain kinds of data such as graphs and geo-spatial queries.</p> | <p>Flexible schema – can store images, audio, video, text (unstructured data), etc.</p> <p>The document model (as with MongoDB) maps to the objects in the application code, making it easier to work with the data.</p> <p>Moreover, ad hoc queries, indexing, and real-time aggregation provide powerful ways to access and analyze your data.</p> |
| Scalability  | <p>RDBMS has challenges in handling huge volumes of data (terabytes and petabytes) despite the availability of RAID (Redundant Array of Independent/ Inexpensive Disks) and data shredding.</p> <p>RDBMS scales up vertically – hardware upgraded as load increases.</p>   | <p>Scales out horizontally – new machines are added as load increases. This means that thousands of cheap commodity machines can provide the required speed and redundancy to support huge volumes of data.</p>                                                                                                                                      |
| Overall cost | <p>Expensive high-end servers – mostly proprietary software with additional licensing cost. In other words, it is expensive monolithic architecture.</p>                                                                                                                   | <p>Open-source software on low-cost commodity machines reduce licensing cost by 50%–80%.</p>                                                                                                                                                                                                                                                         |
| Performance  | <p>Performance degrades as concurrent users increase.</p>                                                                                                                                                                                                                  | <p>Multiple nodes to process multiple requests.</p>                                                                                                                                                                                                                                                                                                  |

| Requirement | Challenges using SQL Databases                   | Solution using NoSQL Databases                                                |
|-------------|--------------------------------------------------|-------------------------------------------------------------------------------|
| Latency     | On-disk processing slows down query performance. | In-memory caching option is available to increase the performance of queries. |

Given the above considerations, NoSQL seems a better fit for scenarios such as e-commerce.

### 6.1.1 NoSQL versus SQL

So, which type of database should be used when? The answer is in the following points:

1. When the requirement is for the data to be consistent throughout – RDBMS.
2. When you experience budgetary constraints and have been asked to make do with low-cost commodity hardware machines – NoSQL.
3. When the data structures are variable – NoSQL.
4. For heavy reads and writes – NoSQL.
5. For event capture and processing – NoSQL.
6. For online stores with complex intelligence engines – NoSQL.
7. When great support is required from vendors – RDBMS.
8. When database maturity is the prime requirement – RDBMS.

#### Picture This

An enterprise “XYZ” has an internal discussion forum for its employees. Employees can create a post, add images and links to external content from the web, audio, video, etc. They can leave a comment on a post, tag their peers and teams, share the post, rate the post, etc. They can create a group, join groups, follow other employees, etc. The Human Resources (HR) department would like answers to the below queries:

1. Which is the most recent blog?
2. Which is (are) the most popular blog(s) – with maximum comments, maximum views?
3. Which is (are) the blog(s) with the highest rating?

**4.** Which is (are) the blog(s) with images?

*Which is/are the most appropriate database/databases to help maintain the data for the enterprise's discussion forum? Justify your answer.*

**Answer:** NoSQL databases are best suited to hold the data for the enterprise's discussion forum. This is because of the following features:

- Ability to deal with heterogeneous data – text, audio, images, video, comments. Graph database helps to store social graph information such as who are the employees who liked the blog, commented on the blog post, did they post comments on similar blog posts, which other blogs have they posted comments on, etc.
- Ability to deal with concurrent access to the same piece of data (read post in our example).
- Scale out architecture to deal with volume (posts and reposts, comments, etc.).
- Low latency – the need is to have the comment on the post reflect in real time or near real time.

Let us get back to our case study of the e-commerce application. Let us look at four main activities:

- 1. Shopping cart:** There are customers who browse/search for products on the e-commerce site, select the products that interest them, place them into the shopping cart with an intention of purchasing.
- 2. User activity log:** The digital footprint of the customer on the e-commerce site is captured.
- 3. Product catalog:** The catalog has details of all the products in the repository.
- 4. Recommendations:** Basis the customer's buying behavior from his past transactions, his interests, and his browsing history, the application is able to recommend products to the customers. This facilitates cross-sell and up-sell.

*Which NoSQL database is best suited for the above-mentioned activities? Will one database suffice or is it a case of polyglot (multiple databases used to take care of the activities)?*

In order to identify the most appropriate NoSQL for the above-mentioned four activities, let us consider two stages:

**Stage I:** This stage is primarily concerned with:

1. Identifying the data model? What kind of data do we have? Is the data text-based, structured, highly related to one another, key-value, etc.?
2. What is the required consistency level? Or what is the tradeoff between consistency and availability?
3. When you look at the operations, are they read/write intensive?

**Stage II:** In this stage, we will map the outcome of stage 1 to the characteristics of few popular NoSQL databases to choose the most suitable one database.

Let us begin with the first component, the shopping cart.

| Parameter        | Business Need                                                                                           | Why not SQL?                                                                                                                                                                                                                                                                               | Why use Key-Value Stores?                                         |
|------------------|---------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| Uniqueness       | Each customer's shopping cart has a set of unique items/products that the customer intends to purchase. | In a relational database which is normalized by default (1NF is the default), you cannot have multiple values for a single customer id. However, it seems to be the requirement in this case. A customer id is required to be mapped to a cart which comprises of multiple items/products. | Key-value stores can be leveraged to represent the shopping cart. |
| High Scalability | The e-commerce site is visited by millions of online customers. The requirement is                      | Relational databases scale up or scale vertically and there is a limit on how much they can scale.                                                                                                                                                                                         | It is possible to scale out cart details onto multiple machines.  |

| Parameter         | Business Need                                                                                                                                                     | Why not SQL?                                                                        | Why use Key-Value Stores?                                                                                |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|
|                   | to manage the shopping cart of each customer.                                                                                                                     |                                                                                     |                                                                                                          |
| Faster Read/Write | Customers would like a quick access to their shopping carts. They may place, replace items/products into their shopping cart till they decide to go for checkout. | Since data is being accessed from a disk, therefore it results in slow performance. | The data can be stored or held in RAM (in-memory cache), hence it is faster than most relational stores. |

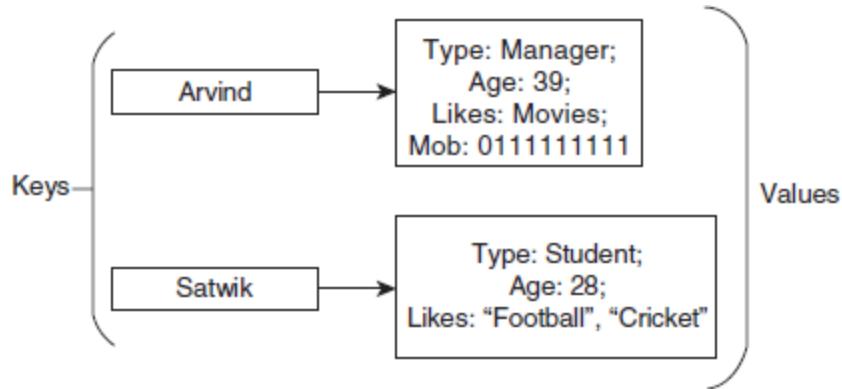
For the above case scenario, the most appropriate database is key–value store.

## 6.2 KEY–VALUE STORE

Figure 6.1 gives a quick recap of the key–value store.

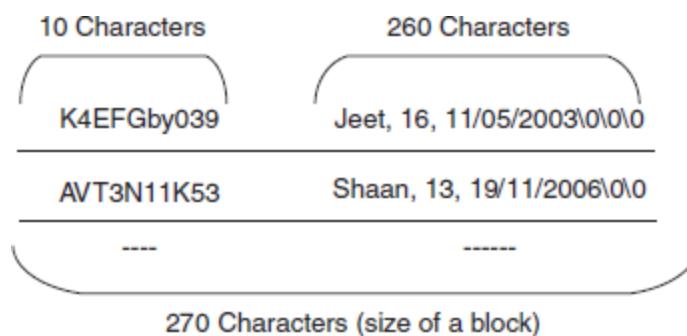
| Key | Value              |
|-----|--------------------|
| K1  | XXX, YYY, ZZZ      |
| K2  | XXX, YYY           |
| K3  | XXX, ZZZ           |
| K4  | XYZ, 3, 01/01/2019 |
| K5  | 3, XYZ, 3213       |

Figure 6.1 Key–value store.



**Figure 6.2** An example of key-value store.

A key–values store stores a collection of key-and-value pairs. A value could be a single value such as customer name or multi-value such as a list of products placed in the shopping cart and identified by a key such as customer ID. Multi-values are separated by delimiters (such as a comma) (refer Figures 6.1 and 6.2). As shown in Figure 6.2, a key “Arvind” is associated with a set of values such as “ “Type: Manager”, age: 39, Likes: movies and Mob: 0111111111”. The key–value pairs are organized into blocks with fixed length (this enables fast traversal between records; refer Figure 6.3).



**Figure 6.3** One single block of key-value store (\0 represents empty/null value).

### 6.2.1 Pros

1. One key – multiple values. Values can be anything (XML, binary, JSON, text, etc.).
2. Easy look-up (search) basis the key.
3. Simple data format makes read and write operations fast.
4. Supports frequent small read/write requests.

5. All the major key-value databases store their data in memory or at least have the feature to store in memory.
6. Very easy to scale. They offer convenient ways of managing nodes.
7. Best fit to store transient data such as user session, user profile, etc.
8. These databases can store large data objects such as images, audio, and video files and provide high performance, flexibility, and scalability.
9. Extremely quick response time (in single-digit millisecond/microsecond).
10. Each key-value pair is unrelated.

### 6.2.2 Limitations

1. Cannot search for items based on the value (values). Searches are only key-based.
2. Does not allow updating part of the value.
3. Normalization-based database relationships are non-existent.
4. Not suitable for complex queries. There the more suitable types are document stores/column family, etc.

### 6.2.3 Use Cases

1. Storing transient attributes in a web application such as shopping cart. Shopping cart items/products are usually temporary, therefore storing them in memory is beneficial. Also, basis the systems requirement, one can set the cart data to expire after a certain amount of time. If need be, the data could be moved to a permanent storage beforehand, thereby allowing the user to continue from where he left off.
2. Session data is keyed by the session ID. Storing sessions is by far the most common use case we have come across. Session data is generally read on every request, so this will save a lot of requests to your main data store or filesystem.
3. Anything with heavy read traffic. Let us say we want to show a collection of statistics to anyone who visits a web page and we have over a million viewers per minute. Instead of running numerous database queries per page load, we could cache the results in memory every minute and then fetch the data from memory for each request. Now that is going to save you some time.

#### 6.2.4 Few Examples of Key–Value NoSQL Databases

1. Amazon Dynamo (it is part of the AWS suite and is proprietary to them)
2. Redis
3. Riak
4. Memcached

In conclusion, key–value stores are light weight, schema-less, relationship-less, and transaction-less data stores.

#### TRY THIS – 1

Find the odd one out with respect to key–value store:

- (a) As key–value stores are NOT normalized, there are NO additional performance costs at runtime.
- (b) Schema-less hence more flexible and allows working with data directly without worrying about first defining the structure.
- (c) Well suited for heavy transactions such as accounting.
- (d) Exhibits improved performance with respect to processing huge volume of unstructured/semi-structured data.

### 6.3 COLUMN FAMILY STORE

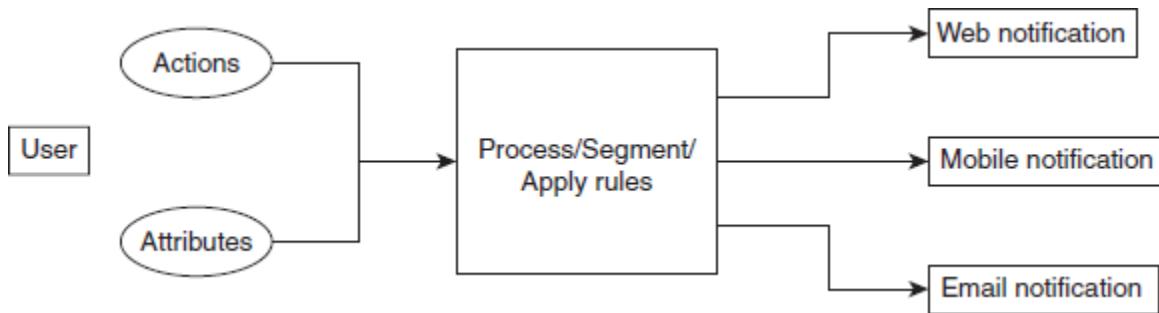
**Problem statement:** Consider a key–value store that stores information about customers. The information stored are customer name, location, purchase\_history, contact\_info, etc. Customer\_id is the key mapped to all details (values) of the customer. The requirement is to generate a list of all customers at a particular location.

*Are key–value stores appropriate for the given scenario?*

The answer is “No”. Key–value stores allow retrieval only by the key (customer\_id). Here the requirement is to list all customers at a particular location. The search should be based on “location”.

Enter column family store...

Almost all e-commerce websites track user activity, their buying behavior, their interests and preferences to leverage opportunities to grow business revenue. Refer [Figure 6.4](#).



**Figure 6.4** E-commerce site process user actions and attributes to leverage business opportunities.

Studying user's behavior and action will help to categorize the users into the following categories:

- 1. Loyal Customers:** Loyal customers are customers who keep tab of the new arrivals and are frequent buyers. Personalized messages, discount coupons, free delivery, etc. can go a long way in retaining them and of course ensure that business keeps growing.
- 2. Repeat Customers:** Repeat customers are customers who keep coming back to the site for their purchases. It will be great for businesses to convert them into loyal customers by welcoming them with personalized mailers each time they visit the site and offer exclusive discount to retain them.
- 3. Couponers:** They shop only during sale period and when coupons are given to them. They can be notified of forthcoming sales, given exclusive coupons to garner quick sales.
- 4. Cart Abandoners:** They are those who place products/items into the cart but their mind is not yet made to go for the purchase. They abandon the cart but may visit the site again. Businesses seek to gain by offering these customers time-bound exclusive discounts.
- 5. Customers with Cart Values Exceeding a Particular Amount:** These are the customers who offer an opportunity to cross-sell and up-sell. For example suppose a customer's cart value is at INR 900 INR. If he/she spends an extra 100 INR, they can get a coupon or will become eligible for a freebie, etc.

**So what are the requirements to track the user activity?**

1. **Requirement 1:** The e-commerce site has thousands and millions of customers. To log their continuous interaction with the system, there is a need of a highly scalable database.
2. **Requirement 2:** Querying by fields other than the key. For example, fetching customer details using *email\_id* which is a non-key attribute (assuming the key attribute is customer id).
3. **Requirement 3:** Retrieving/updating only part of the value. For example, fetching or changing only the customer's *age* and *city/address* among other details.

**Why not SQL?**

1. Millions of customers interacting with the e-commerce system cause heavy logging activity. As RDBMS do not scale well, they fail to meet requirement 1.
2. Also, due to their single node model, these systems fail when overloaded with too many writes.

**Why not key-value store?**

1. Key-value stores can easily meet requirement 1 owing to their ability to scale easily. Key-value stores such as Redis, Memcached, etc. allow for in-memory storage, thereby allowing for faster performance. They are suitable for read and write intensive operations as well.
2. Key-value stores do not allow querying by value or updating only part of the value. Hence, they fail to meet requirements 2 and 3.

**Why column family store?**

1. Allows creation of indexes on non-key values.
2. Write-intensive – allows heavy writes.
3. Allows updating any field.
4. Allows searches based on non-key values.
5. Used to store sparse data (not every row will have every column; if a row does not have a column, no data is stored, making data storage compact and efficient).

**TRY THIS – 2**

---

Choose the most appropriate use of column store:

- (a) Used for session management
- (b) Used to represent highly related data
- (c) Used to store sparse data
- (d) Used for OLTP

**Note:** For frequently updating data and searching by primary key or indexed keys, column families work very well.

There could be queries that make use of multiple fields. To enhance read performance, it may be required to create indexes on many fields. However, creating indexes on multiple fields is not a good practice. It is also a very costly operation. In the light of the above considerations, column family stores are not recommended.

For case scenarios where millions of customers are frequently searching for products based on various criteria (read heavy workloads), document-oriented databases are most suitable. This category of NoSQL databases will be discussed next.

## 6.4 DOCUMENT STORE

---

**Problem statement:** Assume a scenario where customers need to search the catalog for products based on various criteria involving multiple fields as specified below:

1. All products with *price ranging between Rs. 1000 and Rs. 15000*.
2. All products by a specific *brand*.
3. All products sold by a particular *retailer*.
4. All products pertaining to a specific *color, size, type*, and so on.

A column family store will not suffice for the above scenario as the search criteria will involve multiple fields.

Enter document store...

Document-oriented stores help with searches involving multiple fields. Take a moment and think about product catalog.

**Why not SQL?**

1. The schema for ProductCatalog would have to represent unique attributes of all products as follows. The example shows only few attributes for “Apparel” and “Book”. “Apparel” is represented by attributes, “ProductID”, “Price”, “Brand”, “Size”, “Fabric”, “Sleeves” and “Collar”. “Book” is represented by attributes, “ProductID”, “Price”, “BookTitle”, “Author”, “Publisher” and “YearofPublication”. Refer [Figure 6.5](#). The common attributes for “Apparel” and “Book” are “ProductID” and “Price”; these are represented by the first block. The second block represents the unique attributes for “Apparel” and the third block represents the unique attributes for “Book”.

| Common attributes |       | Apparel's attributes |      |        |         |        |           |        | Book's attributes |                   |  |
|-------------------|-------|----------------------|------|--------|---------|--------|-----------|--------|-------------------|-------------------|--|
| ProductID         | Price | Brand                | Size | Fabric | Sleeves | Collar | BookTitle | Author | Publisher         | YearofPublication |  |

**Figure 6.5** Schema for product catalog.

2. This results in hundreds of columns and too many NULL values for columns not relevant to the product. For example, for a book, all fields defined under “Apparel’s attributes” will have NULL values.
3. Also, a single change in attribute requires altering the entire schema. Therefore, RDBMS is inefficient for the given scenario.

### **Why not key–value store?**

1. Customers often search for products based on various criteria, for example, *book with “Publisher” as “Wiley India”, “YearofPublication” as “2019”, etc.*
2. You cannot use *Publisher* or *YearofPublication* while looking up details using key–value stores (only *ProductID* can be used).

### **Why not column store?**

1. Column family stores do not allow querying on multiple fields, for example, *searching for “Women Shirts” whose price is less than INR 500*.
2. Moreover, these databases have limited querying options, for example, operation for *counting the number of books against each publication* is not available.

### **Why document store?**

*Pros:*

1. Limitless throughput.
2. Low latency.
3. Rich query support – allows index to be created on non-key fields. Allows queries to use multiple fields.
4. Predictable performance.
5. Heavy reads.
6. Aggregations are easily performed.

**Limitations:** Cannot represent complex relationships amongst data.

### TRY THIS – 3

**Problem statement:** There is a massive scale application that needs rich queries over a flexible data model, predictable performance, limitless throughput, and/or global distribution to provide low-latency access to any number of regions over a single dataset.

Choose the most appropriate data model for the given scenario.

- (a) SQL database
- (b) Key-value stores
- (c) Column-family
- (d) Document-oriented

## 6.5 GRAPH STORE

**Problem statement:** Let us assume that an online retailer wishes to do the following:

1. Keep track of the products you have purchased and suggest which of your friends have liked the same product.
2. Recommend which other product can be purchased along with the product you have chosen.
3. Show the following once you show interest in a product. Assume that you are interested in “Lee Cooper Women Stylish Top”.
  - (a) “Customers who viewed this item also viewed”
  - (b) “Sponsored Products related to this item”

- (c) “Your recently viewed items and featured recommendations”
- (d) “Your browsing history”

Will document-oriented databases be apt for the given scenario?

The first requirement needs to link your orders to the orders of other related customers. The second requirement needs to maintain information about related products. The third requirement needs to link your interest with the customers who had similar interests and also maintain information about related products.

All of the above point towards highly inter-related data. Relationship between data is not something that is supported by document-oriented databases. Hence, for the given requirements, document-oriented databases are not suitable.

Solution for this recommendation use case is provided by graph databases that we discuss next.

Consider the **recommendations** component in an e-commerce system.

### ***Business Need***

1. Track related products like Kindle and Kindle accessories such as smart flip case cover, USB charger, etc.
2. Relate millions of customers and the products they purchase.

### ***Why not SQL?***

1. You need to join thousands of related products, for example, *recommending purchase of smart flip case cover while purchasing a “All new Kindle paperwhite 10<sup>th</sup> generation”*.
2. SQL Joins have always been known to be an overhead and a very time-consuming operation that slows down query processing.
3. Similar performance bottlenecks are experienced while maintaining relationships between millions of customers and the products each one purchases.

Hence SQL/relational database is not the right solution for this scenario.

### ***Why not key-value stores, column-families, or document databases?***

1. Data required for making appropriate recommendations are highly related. For example, recommending products to customers based on their past purchase history or browsing patterns.
2. None of these databases (key-value stores, column families, or document databases) support complex relationships, hence are not suitable to meet

our needs.

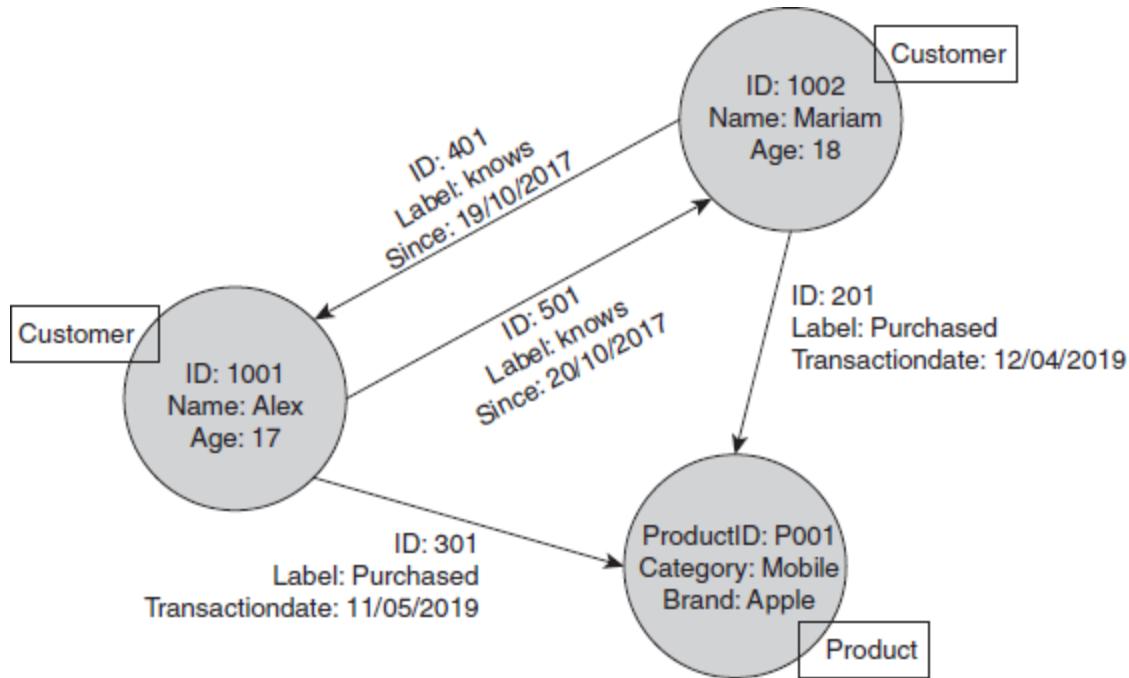
### Why use graph-based databases?

1. These databases use simple nodes and edges to store and retrieve highly interrelated data, for example, customer and product.
2. Graph traversals are much faster compared to SQL joins.

In the given scenario, use graph databases as described below:

1. Users and Products are represented as nodes.
2. Purchased relationships are represented as edges.

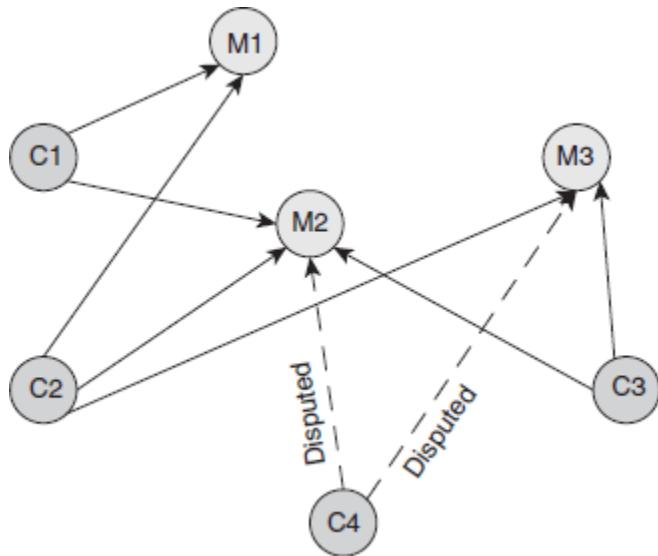
Refer [Figure 6.6](#). Customer “Alex” knows customer “Mariam” since “19/10/2017”. Both customers “Alex” and “Mariam” purchased product “ProductID: P001” of Brand “Apple”.



**Figure 6.6** Users and products represented as nodes and purchased relationships as edges.

Graph databases are also used for fraud detection wherein

1. *Customers* are represented as nodes.
2. *Transactions* are represented as edges.
3. *Transaction* paths that are not related to any *customer* are identified as frauds (refer [Figure 6.7](#)).



**Figure 6.7** Visualizing graph data for fraud detection.

In [Figure 6.7](#), nodes C1, C2, C3, and C4 are customer nodes and nodes M1, M2, and M3 are merchant nodes. The edges connecting the customer nodes with merchant nodes represent transactions. The dashed line represents transactions that are “disputed”. If you take a close look at [Figure 6.7](#), it is way simpler to understand the transactions. We have one customer C4 with fraudulent transactions made at two merchant outlets, M2 and M3.

### ***Limitations of graph-based databases***

1. These databases are inappropriate for transactional data like financial accounting. Relational databases are the most popular and best suited for transactional data owing to their adherence to ACID (Atomicity, Consistency, Isolation and Durability) properties.
2. They do not scale out horizontally.
3. There is difficulty in performing aggregations like sum and max efficiently.
4. It requires one to learn a new query language like CYPHER.
5. There are fewer vendors to choose from, so harder to get technical support.

### **TRY THIS – 4**

Which of the given scenario is most appropriate for using graph-based databases?

- (a) Calculating average income.

- (b) Building a shopping cart.
- (c) Storing structured product information.
- (d) Describing how a user got from point A to point B.

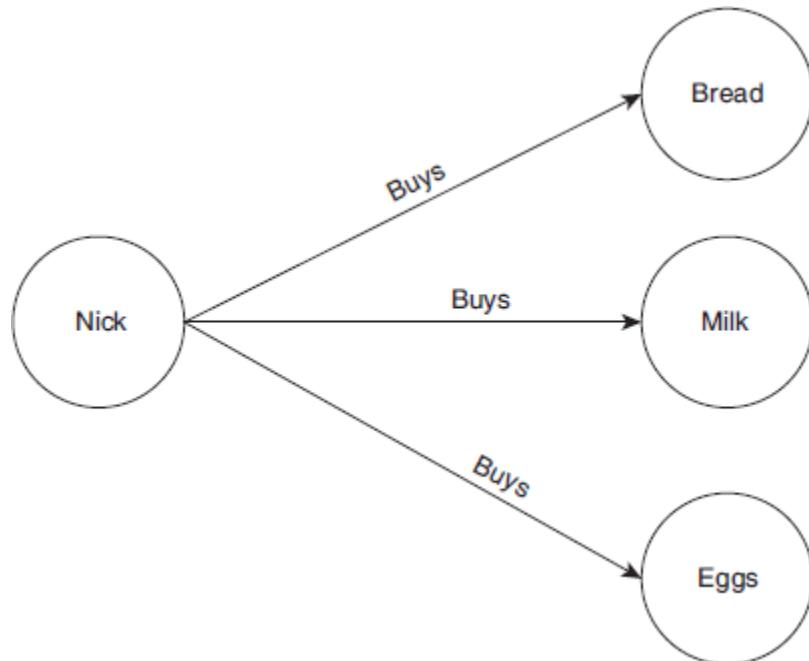
### **Why graph stores?**

Let us look at a few application areas of graph stores:

**1. Market Basket Analysis:** The primary objective of market basket analysis is to recommend items to users based on previous transactions by other users. By simply storing the data in the form of a graph, we leverage the implicitly defined relationships in the data to generate recommendations by using a simple query. This operation is so simple that recommendations can be generated real-time on demand.

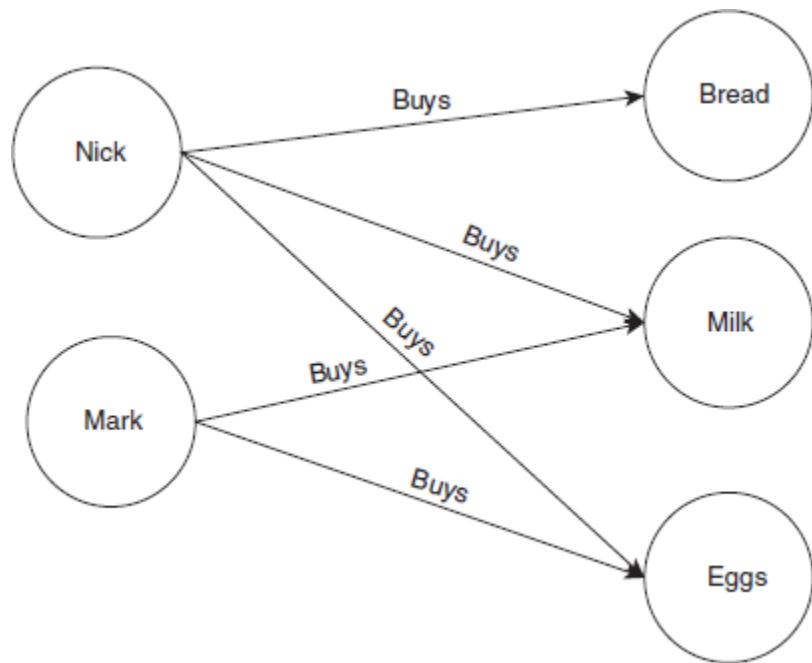
Consider the following customer transactions:

- Nick buys bread, milk, and eggs. Nick, a customer is stored as a node in the graph. Items bread, milk, and eggs are also stored as nodes in the graph. The relationship “BUYS” associates customer node “Nick” to item nodes “Bread”, “Milk”, and “Eggs”. Refer [Figure 6.8](#).



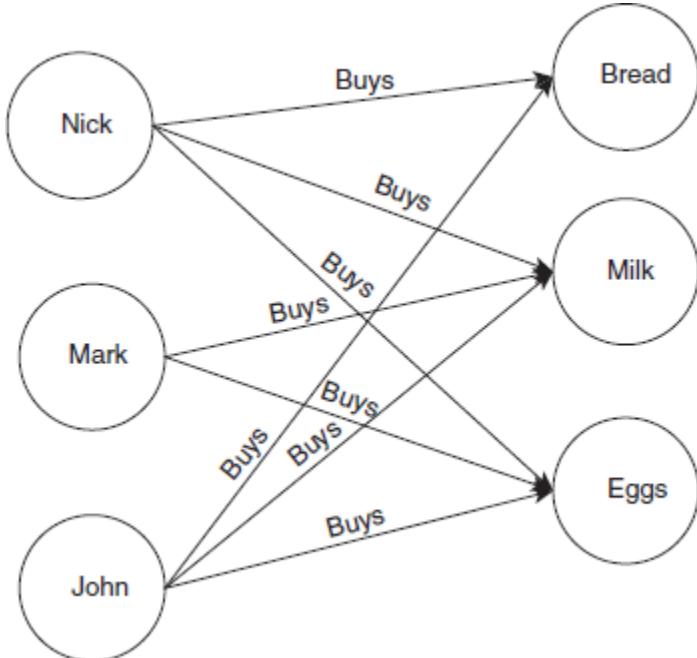
**Figure 6.8** Customer “Nick” bought items “Bread”, “Milk”, and “Eggs”.

- Mark buys milk and eggs. Refer [Figure 6.9](#).



**Figure 6.9** Customer “Mark” bought items “Milk” and “Eggs”.

- John buys milk, eggs, and bread. Refer [Figure 6.10](#).



**Figure 6.10** Customer “John” bought items “Bread”, “Milk”, and “Eggs”.

- George, a new user just bought milk. We now need to generate recommendations for him.

To generate recommendations for George, we traverse the graph in the following way:

- We start from the customer node “George” and visit the item he just bought, “Milk”.
- From the “Milk” node, we traverse to all the other customer nodes that have the “BUYS” relationship with it – essentially finding other users who have also bought milk.
- Now that we have other customers who have bought milk, we just need to traverse to the other items these customers also bought. The strength of these recommendations has a direct correlation with how many “other” customers bought these “other” items. Hence, that will be the parameter we can use to rank the recommendations.

## 2. Real Time recommendations:

Walmart uses graph database such as Neo4j to serve up real time recommendations to its customers on its websites. They use Neo4j to study their customers’ shopping behavior and the relationship between customers and products to provide product recommendations in real time.

- 3. Banking Fraudulent Transactions** – Graphs can be used to find unusual patterns helping in mitigating Fraudulent transactions.
- 4. Supply Chain** – Graphs help in identifying optimum routes for your delivery trucks and in identifying locations for warehouses and delivery centers.

### **Pros**

Deals beautifully with relationships. Graph-based store is very suitable for representing highly interrelated data.

### **Cons**

1. Scaling out is a concern.
2. Aggregations are not easy.
3. Few vendors in the market for graph-based stores. Therefore, very less technical support is available.
4. This requires the knowledge of a new query language such as cipher.
5. It is not suitable for transactional data.

## 6.6 EXAMPLES OF NoSQL DATABASES

---

### 6.6.1 Key–Value Stores

1. **Redis:** Redis is distributed, in memory key–value store. It is most appropriately called as data structure store owing to its support for lists, strings, maps, sets, bitmaps, streams, spatial indexes, etc.
2. **Memcached:** Memcached is an open-source, general purpose, distributed in-memory key–value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. Memcached is NOT:
  - A persistent data store
  - A database
  - Fault tolerant
  - Highly available
3. **Riak:** Riak key–value store is
  - Open source
  - Distributed
  - Fault tolerant
  - AvailableIt is available as an open-source version, enterprise version, and a cloud version.
4. **DynamoDB:** DynamoDB is a key–value and document database. It is fully managed proprietary offering from Amazon and is offered as a part of the Amazon Web Services portfolio.
5. **Project Voldemort:** Project Voldemort was designed as a distributed key–value store. It was used by LinkedIn. It is named after the fictional Harry Potter villain Lord Voldemort.

### 6.6.2 Document Stores

1. **MongoDB:** MongoDB is a document data store that stores data in the form of JSON (Java Script Object Notation).
2. **CouchDB:** CouchDB is a document-oriented NoSQL database. It is written in Erlang language. It stores data in JSON document, uses Java Scripts as a query language using Map Reduce, and HTTP for an API.

3. **Couchbase:** Couchbase is a new version of Membase. It is an open-source, distributed, document-oriented database.
4. **MarkLogic:** MarkLogic Sever is a document-oriented database developed by MarkLogic Corporation. It is a multi-nodal NoSQL database.

### 6.6.3 Column Stores

1. **Apache HBase:** Apache HBase is an open-source, distributed column-oriented/key-value data store that runs atop Hadoop Distributed File System. It is modeled after Google's Big Table. It was developed by Apache Software foundation and is written in Java.
2. **Apache Cassandra:** Apache Cassandra is a column-family NoSQL database.

### 6.6.4 Graph Stores

1. **Neo4j:** Neo4j is a graph NoSQL database.
2. **OrientDB:** OrientDB is
  - Open source
  - NoSQL
  - Has features of graph as well as document database
  - Written in Java
  - Exceedingly fast
3. **InfiniteGraph:** InfiniteGraph is a distributed graph database.

## REMEMBER ME



- 
1. NoSQL databases support dynamic schema and the data can be stored in many different ways – Key-value pairs, document store, column store, or organized as graph based database.
  2. NoSQL databases are horizontally scalable, which means that they can handle increased traffic simply by adding more servers to the database.
  3. NoSQL databases have numerous benefits, including lower cost, open-source availability, and easier scalability, which makes it an appealing option for anyone thinking about integrating in Big Data.
  4. NoSQL databases have the ability to deal with heterogeneous data – text, audio, images, video, and comments.

5. A key-values store stores a collection of key-and-value pairs. A value could be a single value such as customer name or multi-value such as a list of products placed in the shopping cart and identified by a key such as customer ID.
6. Column family store is used to store sparse data (not every row will have every column; if a row does not have a column, no data is stored, making data storage compact and efficient).
7. Document store has a rich query support – allows index to be created on non-key fields. Allows queries to use multiple fields.
8. Graph databases use simple nodes and edges to store and retrieve highly interrelated data, for example, customer and product. Graph traversals are much faster compared to SQL joins.

## QUESTION ME

---

1. Is Couchbase the same as CouncillDB?
2. How is Couchbase related to Membase?
3. Which is better – Redis or Memcached?
4. Are all relational databases consistent and available?
5. What is a multi-nodal NoSQL database?
6. Explain “polyglot persistence” in NoSQL.
7. Does NoSQL interact with Oracle database?
8. When should you use a NoSQL database instead of relational database?
9. Give three differences between relational and NoSQL databases.
10. Can relational database be multi-nodal?

## REFERENCE ME

---

1. <https://www.mongodb.com/>
2. <https://en.wikipedia.org/wiki/MongoDB>
3. <https://neo4j.com/>
4. <https://en.wikipedia.org/wiki/Neo4j>
5. [https://en.wikipedia.org/wiki/Key-value\\_database](https://en.wikipedia.org/wiki/Key-value_database)
6. [https://en.wikipedia.org/wiki/Column-oriented\\_DBMS](https://en.wikipedia.org/wiki/Column-oriented_DBMS)
7. [https://en.wikipedia.org/wiki/Graph\\_database](https://en.wikipedia.org/wiki/Graph_database)
8. <https://neo4j.com/developer/graph-db-vs-rdbms/>

# ANSWERS

---

## Try This – 1

- (c) Well suited for heavy transactions such as accounting.

## Try This – 2

- (c) Used to store sparse data.

**Reason:** The other options are invalid due to the following reasons:

- For OLTP, relational database is the most appropriate choice.
- For session management, key–value store is the most appropriate choice.
- For highly related data, graph store is the most suitable.

## Try This – 3

- (d) Document-oriented. These databases are best suited because of their support for

**Reason:** The other options are invalid due to the following reasons:

- *Flexible data model:* Relational or SQL databases are not suitable.
- *Rich queries:* Key–value stores and column-family stores have limited query support, whereas document-oriented databases provide rich querying options.

## Try This – 4

- (d) Describing how a user got from point A to point B.

**Reason:** The other options are invalid due to the following reasons:

- Calculating average income: This can be easily computed by any SQL/relational database.
- Building a shopping cart: Key–value store is the most suitable for this transient shopping cart data.
- Storing structured product information: Either document-oriented or column-family store is the most apt for this.

*OceanofPDF.com*



## **Annexure A – Project 1 in MongoDB Database**

“XYZ” is a large size enterprise with over 2,00,000 employees. They quickly realized that to stay ahead of their competition they will have to invest in learning and development of their employees. To this effect, they entered into a contract with a new age learning partner by the name of “LNZ”. They have purchased licenses to the following:

- Learning content.
- Hands-on environment.
- Certification vouchers.

They have purchased 5000 licenses to the learning content with a validity period of one year. The learning content are of many types, including e-books, audio podcast, video podcast, voice over presentations, etc. The contract period is from 1 January 2019 to 31 December 2019. This implies that if the license is granted on 1 January, an employee gets a year to consume the content. Moreover, if the license is granted to the employee on 1 April, even then the license will expire on 31 December 2019.

The hands-on environment will help the employees to master the technology with adequate hands-on practice. They have 200 licenses. There are three types of learning environment – Windows based, Linux based and Ubuntu based. Each license has a validity of 1 month from the date of granting access to the employee. The contract is valid from 1 March 2019 to 29 February 2020. If an employee is given access on 1 March 2019, the validity is for one month from 1 March 2019 (i.e., it will be valid till 31 March 2019). However, if the access is granted on 29 February 2020, then the validity is till 28 March 2020.

The certification vouchers will enable employees to be externally certified. The enterprise has bought 2000 licenses. Each license is twofold, which implies that a voucher entitles an employee to two types of certifications: (a) Associate and (b) Professional. Each type allows the employee a maximum of two attempts to qualify. The validity is for 1 year, from the date of registration by an employee.

In this assignment, you will be working on the sample data sets of the “LearningManagementSystem” System.

1. Create a database named “LearningManagementSystem” and insert the given collections.
2. Write a query to display the number of resource types available under contract type “Learning Content” (Hint: use “Resources” table/collection and contract id = C101).
3. Write a query to display the number of hands-on playground available under contract type “HandsOn Environment” (Hint: use “Resources” table/collection and contract id = C102).
4. List the number of users who have been given access to all the three contract types “Learning Content”, “HandsOn Environment” and “Certification Vouchers”. (Use the “Users” table/collection).
5. Which course has been accessed  $\geq 3$  times (Use “Learning\_Content” table/collection).
6. Which “HandsOn Playground” has been accessed the maximum number of times – (Windows, Linux or Ubuntu). (Hint: use “HandsOn\_Playground” table/collection).
7. How many cleared the certifications in their first attempt? (Use “Certification\_Vouchers” table/collection).
8. How many users have been given the “certification vouchers”? (Hint: use “Users” table/collection).
9. Get details (Contract start date, contract end date, # of licenses etc.) of each contract type. (Hint: use “Contracts” table/collection).
10. Is “Video Podcast” available for “Introduction to AI” course? (Hint: Use “Courses” table/collection).
11. Is “Audio Podcast” available for “Introduction to ML” course? (Hint: Use “Courses” table/collection).
12. What types of resources are available under “Learning Content”? (hint: “E-books”, “Audio Podcast”, “Video Podcast”, “Voice over presentations”, etc. are available).

## Contracts

| Contract_ID | Contract_Type          | Contract_St_Dt | Contract_End_Dt | #licenses |
|-------------|------------------------|----------------|-----------------|-----------|
| C101        | Learning Content       | 01-Jan-2019    | 31-Dec-2019     | 5000      |
| C102        | HandsOn Environment    | 01-Mar-2019    | 29-Feb-2020     | 500       |
| C103        | Certification Vouchers | 01-Apr-2019    | 31-Mar-2020     | 2000      |

## Resources

| Contract_ID | Resource_ID | Resource_Type            |
|-------------|-------------|--------------------------|
| C101        | C101-1      | E Books                  |
| C101        | C101-2      | Audio Podcast            |
| C101        | C101-3      | Video Podcast            |
| C101        | C101-4      | Voice Over presentations |
| C101        | C101-5      | Live Class over Web      |
| C101        | C101-6      | Events over Web          |
| C102        | C102-1      | Windows Playground       |
| C102        | C102-2      | Linux Playground         |
| C102        | C102-3      | Ubuntu Playground        |
| C103        | C103-1      | Associate                |
| C103        | C103-2      | Professional             |

## Users

| User_ID | User_EmailID     | Contract_ID | Access_Grant_Dt | Access_Expiry_Dt |
|---------|------------------|-------------|-----------------|------------------|
| U101    | Bob.D@xyz.com    | C101        | 02-Jan-2019     | 31-Dec-2019      |
| U101    | Bob.D@xyz.com    | C102        | 09-Mar-2019     | 08-Apr-2019      |
| U101    | Bob.D@xyz.com    | C103        | 25-Apr-2019     | 24-Apr-2020      |
| U102    | Andrew.M@xyz.com | C102        | 11-May-2019     | 10-June-2019     |
| U102    | Andrew.M@xyz.com | C103        | 26-Apr-2019     | 25-Apr-2020      |

| User_ID | User_EmailID       | Contract_ID | Access_Grant_Dt | Access_Expiry_Dt |
|---------|--------------------|-------------|-----------------|------------------|
| U103    | Christie.M@xyz.com | C101        | 9-Jan-2019      | 31-Dec-2019      |
| U104    | Freddy.A@xyz.com   | C101        | 02-Jan-2019     | 31-Dec-2019      |
| U104    | Freddy.A@xyz.com   | C102        | 11-Mar-2019     | 10-Apr-2019      |
| U104    | Freddy.A@xyz.com   | C103        | 25-Apr-2019     | 24-Apr-2020      |
| U105    | Maratha.x@xyz.com  | C102        | 12-Apr-2019     | 11-May-2019      |
| U105    | Maratha.x@xyz.com  | C103        | 12-Apr-2019     | 11-Apr-2020      |
| U106    | Mabel.z@xyz.com    | C101        | 11-Jan-2019     | 31-Dec-2019      |
| U107    | Alex.D@xyz.com     | C103        | 11-May-2019     | 10-May-2020      |
| U108    | Turing.S@xyz.com   | C102        | 20-Jan-2019     | 19-Feb-2019      |
| U108    | Turing.S@xyz.com   | C103        | 11-May-2019     | 10-May-2020      |
| U109    | Zidane.r@xyz.com   | C101        | 01-Feb-2019     | 31-Dec-2019      |
| U109    | Zidane.r@xyz.com   | C102        | 09-Apr-2019     | 08-May-2019      |
| U109    | Zidane.r@xyz.com   | C103        | 25-Apr-2019     | 24-Apr-2020      |

## Learning\_Content

| User_ID | Resource_ID | Course_ID | Learning_Hours | Last_Active |
|---------|-------------|-----------|----------------|-------------|
| U101    | C101-1      | Cr_1011   | 00:10:10       | 02-Jan-2019 |
| U101    | C101-1      | Cr_1012   | 00:34:00       | 06-Jan-2019 |
| U101    | C101-3      | Cr_1019   | 01:12:30       | 09-Jan-2019 |
| U103    | C101-1      | Cr_1011   | 10:02:20       | 13-May-2019 |
| U103    | C101-3      | Cr_1019   | 02:45:00       | 02-Jun-2019 |
| U104    | C101-3      | Cr_1019   | 02:30:45       | 07-Jun-2019 |
| U104    | C101-3      | Cr_1019   | 04:30:00       | 09-Jul-2019 |
| U103    | C101-3      | Cr_10110  | 03:10:40       | 20-Jul-2019 |
| U101    | C101-1      | Cr_1011   | 01:00:30       | 03-Aug-2019 |
| U106    | C101-5      | Cr_10113  | 03:15:00       | 05-Aug-2019 |
| U109    | C101-5      | Cr_10114  | 03:16:00       | 05-Aug-2019 |
| U109    | C101-5      | Cr_10114  | 02:30:00       | 07-Aug-2019 |

## HandsOn\_Playground

| User_ID | Resource_ID | Course_ID | Learning_Hours | Last_Active |
|---------|-------------|-----------|----------------|-------------|
| U101    | C102-1      | Cr_1021   | 01:02:00       | 15-Mar-2019 |
| U101    | C102-1      | Cr_1022   | 01:30:00       | 19-Mar-2019 |
| U101    | C102-1      | Cr_1022   | 00:30:00       | 02-Apr-2019 |
| U102    | C102-1      | Cr_1023   | 02:20:00       | 15-May-2019 |
| U102    | C102-1      | Cr_1023   | 00:45:00       | 19-May-2019 |
| U104    | C102-3      | Cr_1027   | 03:00:00       | 15-Mar-2019 |
| U104    | C102-3      | Cr_1027   | 02:25:00       | 19-Mar-2019 |
| U102    | C102-2      | Cr_1025   | 01:50:00       | 02-Jun-2019 |
| U101    | C102-1      | Cr_1022   | 02:30:00       | 04-Apr-2019 |
| U108    | C102-3      | Cr_1026   | 04:40:00       | 26-Jan-2019 |
| U109    | C102-2      | Cr_1024   | 02:55:00       | 15-Apr-2019 |
| U109    | C102-2      | Cr_1024   | 01:45:00       | 25-Apr-2019 |

## Certification\_Vouchers

| User_ID | Resource_ID | Course_ID | Attempt | Date_of_Exam | Qualified |
|---------|-------------|-----------|---------|--------------|-----------|
| U101    | C103-1      | Cr_1031   | 1       | 01-Jun-2019  | N         |
| U101    | C103-2      | Cr_1032   | 1       | 01-Sep-2019  | N         |
| U101    | C103-1      | Cr_1031   | 2       | 31-Oct-2019  | Y         |
| U102    | C103-2      | Cr_1032   | 1       | 01-Jul-2019  | N         |
| U102    | C103-2      | Cr_1032   | 2       | 01-Sep-2019  | N         |
| U104    | C103-1      | Cr_1035   | 1       | 01-Nov-2019  | N         |
| U104    | C103-1      | Cr_1035   | 2       | 02-Jan-2020  | Y         |
| U102    | C103-1      | Cr_1031   | 1       | 01-Nov-2019  | Y         |
| U101    | C103-2      | Cr_1032   | 2       | 02-Dec-2019  | Y         |
| U108    | C103-2      | Cr_1034   | 1       | 11-Jun-2019  | Y         |
| U109    | C103-2      | Cr_1034   | 1       | 11-Aug-2019  | Y         |
| U109    | C103-1      | Cr_1035   | 1       | 20-Nov-2019  | Y         |

| User_ID | Resource_ID | Course_ID | Attempt | Date_of_Exam | Qualified |
|---------|-------------|-----------|---------|--------------|-----------|
| U105    | C103-1      | Cr_1035   | 1       | 30-sep-2019  | Y         |

## Courses

| Resource_ID | Course_ID | Course_Name                    | Duration_in_Hours |
|-------------|-----------|--------------------------------|-------------------|
| C101-1      | Cr_1011   | Introduction to BI             | 3                 |
| C101-1      | Cr_1012   | Introduction to Data Warehouse | 4.5               |
| C101-1      | Cr_1013   | Introduction to AI             | 3                 |
| C101-1      | Cr_1014   | Introduction to ML             | 6                 |
| C101-2      | Cr_1015   | Introduction to BI             | 2.5               |
| C101-2      | Cr_1016   | Introduction to AI             | 3                 |
| C101-2      | Cr_1017   | Introduction to ML             | 6                 |
| C101-3      | Cr_1018   | Introduction to BI             | 3                 |
| C101-3      | Cr_1019   | Introduction to AI             | 4.5               |
| C101-3      | Cr_10110  | Introduction to ML             | 6                 |
| C101-4      | Cr_10111  | Introduction to AI             | 6                 |
| C101-4      | Cr_10112  | Introduction to ML             | 6                 |
| C101-5      | Cr_10113  | Introduction to AI             | 6                 |
| C101-5      | Cr_10114  | Introduction to ML             | 6                 |
| C101-6      | Cr_10115  | Introduction to AI             | 6                 |
| C101-6      | Cr_10116  | Introduction to ML             | 6                 |
| C102-1      | Cr_1021   | R setup                        | 32                |
| C102-1      | Cr_1022   | Python setup                   | 40                |
| C102-1      | Cr_1023   | Jaspersoft setup               | 24                |
| C102-2      | Cr_1024   | R setup                        | 32                |
| C102-2      | Cr_1025   | Python setup                   | 40                |
| C102-3      | Cr_1026   | Pentaho                        | 40                |
| C102-3      | Cr_1027   | Talend                         | 40                |
| C103-1      | Cr_1031   | Tableau Associate              | 48                |

| Resource_ID | Course_ID | Course_Name                       | Duration_in_Hours |
|-------------|-----------|-----------------------------------|-------------------|
| C103-2      | Cr_1032   | Tableau Professional              | 72                |
| C103-1      | Cr_1033   | Cognos Developer Associate        | 48                |
| C103-2      | Cr_1034   | Cognos Developer Professional     | 72                |
| C103-1      | Cr_1035   | Informatica Designer Associate    | 48                |
| C103-2      | Cr_1036   | Informatica Designer Professional | 72                |

*OceanofPDF.com*



## **Annexure B – Project 2 in MongoDB Database**

“XYZ” is a food mart. They stock more than 75 types of cereals from various manufacturers like Kellogg’s, Quaker Oats, General Mills, etc. They have maintained a database called CerealAnalysis to store ingredient details for each cereal, which will help them suggest the best buy for their customers as per their requirements.

In this project, you will be working on the sample data set of “Cereals”.

1. Create a database named “CerealAnalysis” and a collection named “Cereals” and then insert the provided data.
2. Write a query to display the count of the number of cereals with a rating of 50% and higher ( $>=50.00$ ).
3. Write a query to display the name of all cereals with a rating of 50% and higher ( $>=50.00$ ).
4. Write a query to display the name of all cereals wherein the number of cups in one serving is in the range of 0.75 to 1 (0.75 to 1.0).
5. Write a query to display the name of all cereals placed on shelf 3.
6. Write a query to display the name of all cereals with the number of calories per serving at 100 and the grams of protein equal to or greater than 3.

7. Write a query to display the name of all cereals from Kellogg's manufacturer.
8. Write a query to determine how many cereals are "hot" and how many cereals are "cold".
9. Write a query to display the name of all cereals whose names begin with a "G".
10. Write a query to determine the name of the cereal which is a "hot" cereal with the number of calories per serving equal to or greater than 100 and the grams of fat is less than 5.
11. Write a query to display the count of cereals with "Corn".
12. Write a query to display the name of the cereal with maximum fiber content.
13. Write a query to display the name of the cereal with minimum number of calories per serving.
14. Write a query to display the name of the cereal which has been rated above 80%, has zero grams of sugar, has zero grams of fat but has very good quantity of fiber.
15. Display the name of all cereals with "Bran" and "Raisins".
16. Write a query to display the count of cereals placed in shelf 1 and 2.

The data is provided under 16 columns in [Table 1](#). The description of each column is provided in [Table 2](#).

**Table 1** Data about cereals

| name                      | mfr | type | calories | protein | fat | sodium | fiber | carbo | sugars | potass | vitamins | shelf | weight | cups | rating    |
|---------------------------|-----|------|----------|---------|-----|--------|-------|-------|--------|--------|----------|-------|--------|------|-----------|
| 100% Bran                 | N   | C    | 70       | 4       | 1   | 130    | 10    | 5     | 6      | 280    | 25       | 3     | 1      | 0.33 | 68.402973 |
| 100% Natural bran         | Q   | C    | 120      | 3       | 5   | 15     | 2     | 8     | 8      | 135    | 0        | 3     | 1      | 1    | 33.983679 |
| All-bran                  | K   | C    | 70       | 4       | 1   | 260    | 9     | 7     | 5      | 320    | 25       | 3     | 1      | 0.33 | 59.425505 |
| All-bran with extra fiber | K   | C    | 50       | 4       | 0   | 140    | 14    | 8     | 0      | 330    | 25       | 3     | 1      | 0.5  | 93.704912 |
| Almond delight            | R   | C    | 110      | 2       | 2   | 200    | 1     | 14    | 8      | -1     | 25       | 3     | 1      | 0.75 | 34.384843 |
| Apple cinnamon cheerios   | G   | C    | 110      | 2       | 2   | 180    | 1.5   | 10.5  | 10     | 70     | 25       | 1     | 1      | 0.75 | 29.509541 |
| Apple jacks               | K   | C    | 110      | 2       | 0   | 125    | 1     | 11    | 14     | 30     | 25       | 2     | 1      | 1    | 33.174094 |
| Basic 4                   | G   | C    | 130      | 3       | 2   | 210    | 2     | 18    | 8      | 100    | 25       | 3     | 1.33   | 0.75 | 37.038562 |
| Bran chex                 | R   | C    | 90       | 2       | 1   | 200    | 4     | 15    | 6      | 125    | 25       | 1     | 1      | 0.67 | 49.120253 |
| Bran flakes               | P   | C    | 90       | 3       | 0   | 210    | 5     | 13    | 5      | 190    | 25       | 3     | 1      | 0.67 | 53.313813 |
| Cap'n crunch              | Q   | C    | 120      | 1       | 2   | 220    | 0     | 12    | 12     | 35     | 25       | 2     | 1      | 0.75 | 18.042851 |
| Cheerios                  | G   | C    | 110      | 6       | 2   | 290    | 2     | 17    | 1      | 105    | 25       | 1     | 1      | 1.25 | 50.764999 |
| Cinnamon toast crunch     | G   | C    | 120      | 1       | 3   | 210    | 0     | 13    | 9      | 45     | 25       | 2     | 1      | 0.75 | 19.823573 |
| Clusters                  | G   | C    | 110      | 3       | 2   | 140    | 2     | 13    | 7      | 105    | 25       | 3     | 1      | 0.5  | 40.400208 |
| Cocoa puffs               | G   | C    | 110      | 1       | 1   | 180    | 0     | 12    | 13     | 55     | 25       | 2     | 1      | 1    | 22.736446 |
| Corn chex                 | R   | C    | 110      | 2       | 0   | 280    | 0     | 22    | 3      | 25     | 25       | 1     | 1      | 1    | 41.445019 |
| Corn flakes               | K   | C    | 100      | 2       | 0   | 290    | 1     | 21    | 2      | 35     | 25       | 1     | 1      | 1    | 45.863324 |
| Corn pops                 | K   | C    | 110      | 1       | 0   | 90     | 1     | 13    | 12     | 20     | 25       | 2     | 1      | 1    | 35.782791 |
| Count chocula             | G   | C    | 110      | 1       | 1   | 180    | 0     | 12    | 13     | 65     | 25       | 2     | 1      | 1    | 22.396513 |
| Cracklin' oat bran        | K   | C    | 110      | 3       | 3   | 140    | 4     | 10    | 7      | 160    | 25       | 3     | 1      | 0.5  | 40.448772 |
| Cream of wheat (Quick)    | N   | H    | 100      | 3       | 0   | 80     | 1     | 21    | 0      | -1     | 0        | 2     | 1      | 1    | 64.533816 |
| Crispix                   | K   | C    | 110      | 2       | 0   | 220    | 1     | 21    | 3      | 30     | 25       | 3     | 1      | 1    | 46.895644 |

| name                       | mfr | type | calories | protein | fat | sodium | fiber | carbo | sugars | potass | vitamins | shelf | weight | cups | rating    |
|----------------------------|-----|------|----------|---------|-----|--------|-------|-------|--------|--------|----------|-------|--------|------|-----------|
| Crispy wheat & raisins     | G   | C    | 100      | 2       | 1   | 140    | 2     | 11    | 10     | 120    | 25       | 3     | 1      | 0.75 | 36.176196 |
| Double chex                | R   | C    | 100      | 2       | 0   | 190    | 1     | 18    | 5      | 80     | 25       | 3     | 1      | 0.75 | 44.330856 |
| Froot loops                | K   | C    | 110      | 2       | 1   | 125    | 1     | 11    | 13     | 30     | 25       | 2     | 1      | 1    | 32.207582 |
| Frosted flakes             | K   | C    | 110      | 1       | 0   | 200    | 1     | 14    | 11     | 25     | 25       | 1     | 1      | 0.75 | 31.435973 |
| Frosted mini-wheats        | K   | C    | 100      | 3       | 0   | 0      | 3     | 14    | 7      | 100    | 25       | 2     | 1      | 0.8  | 58.345141 |
| Fruit & fibre dates        |     |      |          |         |     |        |       |       |        |        |          |       |        |      |           |
| Fruitful bran              | K   | C    | 120      | 3       | 0   | 240    | 5     | 14    | 12     | 190    | 25       | 3     | 1.33   | 0.67 | 41.015492 |
| Fruity pebbles             | P   | C    | 110      | 1       | 1   | 135    | 0     | 13    | 12     | 25     | 25       | 2     | 1      | 0.75 | 28.025765 |
| Golden crisp               | P   | C    | 100      | 2       | 0   | 45     | 0     | 11    | 15     | 40     | 25       | 1     | 1      | 0.88 | 35.252444 |
| Golden grahams             | G   | C    | 110      | 1       | 1   | 280    | 0     | 15    | 9      | 45     | 25       | 2     | 1      | 0.75 | 23.804043 |
| Grape nuts flakes          | P   | C    | 100      | 3       | 1   | 140    | 3     | 15    | 5      | 85     | 25       | 3     | 1      | 0.88 | 52.076897 |
| Grape-nuts                 | P   | C    | 110      | 3       | 0   | 170    | 3     | 17    | 3      | 90     | 25       | 3     | 1      | 0.25 | 53.371007 |
| Great grains pecan         | P   | C    | 120      | 3       | 3   | 75     | 3     | 13    | 4      | 100    | 25       | 3     | 1      | 0.33 | 45.811716 |
| Honey graham oh's          | Q   | C    | 120      | 1       | 2   | 220    | 1     | 12    | 11     | 45     | 25       | 2     | 1      | 1    | 21.871292 |
| Honey nut cheerios         | G   | C    | 110      | 3       | 1   | 250    | 1.5   | 11.5  | 10     | 90     | 25       | 1     | 1      | 0.75 | 31.072217 |
| Honey-comb                 | P   | C    | 110      | 1       | 0   | 180    | 0     | 14    | 11     | 35     | 25       | 1     | 1      | 1.33 | 28.742414 |
| Just right crunchy nuggets | K   | C    | 110      | 2       | 1   | 170    | 1     | 17    | 6      | 60     | 100      | 3     | 1      | 1    | 36.523683 |
| Just right fruit & nut     | K   | C    | 140      | 3       | 1   | 170    | 2     | 20    | 9      | 95     | 100      | 3     | 1.3    | 0.75 | 36.471512 |
| Kix                        | G   | C    | 110      | 2       | 1   | 260    | 0     | 21    | 3      | 40     | 25       | 2     | 1      | 1.5  | 39.241114 |

| name                      | mfr | type | calories | protein | fat | sodium | fiber | carbo | sugars | potass | vitamins | shelf | weight | cups | rating    |
|---------------------------|-----|------|----------|---------|-----|--------|-------|-------|--------|--------|----------|-------|--------|------|-----------|
| Life                      | Q   | C    | 100      | 4       | 2   | 150    | 2     | 12    | 6      | 95     | 25       | 2     | 1      | 0.67 | 45.328074 |
| Lucky charms              | G   | C    | 110      | 2       | 1   | 180    | 0     | 12    | 12     | 55     | 25       | 2     | 1      | 1    | 26.734515 |
| Maypo                     | A   | H    | 100      | 4       | 1   | 0      | 0     | 16    | 3      | 95     | 25       | 2     | 1      | 1    | 54.850917 |
| Muesli raisins & almonds  | R   | C    | 150      | 4       | 3   | 95     | 3     | 16    | 11     | 170    | 25       | 3     | 1      | 1    | 37.136863 |
| Muesli raisins & pecans   | R   | C    | 150      | 4       | 3   | 150    | 3     | 16    | 11     | 170    | 25       | 3     | 1      | 1    | 34.139765 |
| Mueslix crispy blend      | K   | C    | 160      | 3       | 2   | 150    | 3     | 17    | 13     | 160    | 25       | 3     | 1.5    | 0.67 | 30.313351 |
| Multi-grain cheerios      | G   | C    | 100      | 2       | 1   | 220    | 2     | 15    | 6      | 90     | 25       | 1     | 1      | 1    | 40.105965 |
| Nut & honey crunch        | K   | C    | 120      | 2       | 1   | 190    | 0     | 15    | 9      | 40     | 25       | 2     | 1      | 0.67 | 29.924285 |
| Nutri-grain almond-raisin | K   | C    | 140      | 3       | 2   | 220    | 3     | 21    | 7      | 130    | 25       | 3     | 1.33   | 0.67 | 40.69232  |
| Nutri-grain wheat         | K   | C    | 90       | 3       | 0   | 170    | 3     | 18    | 2      | 90     | 25       | 3     | 1      | 1    | 59.642837 |
| Oatmeal raisin crisp      | G   | C    | 130      | 3       | 2   | 170    | 1.5   | 13.5  | 10     | 120    | 25       | 3     | 1.25   | 0.5  | 30.450843 |
| Post nat. Raisin bran     | P   | C    | 120      | 3       | 1   | 200    | 6     | 11    | 14     | 260    | 25       | 3     | 1.33   | 0.67 | 37.840594 |
| Product 19                | K   | C    | 100      | 3       | 0   | 320    | 1     | 20    | 3      | 45     | 100      | 3     | 1      | 1    | 41.50354  |
| Puffed rice               | Q   | C    | 50       | 1       | 0   | 0      | 0     | 13    | 0      | 15     | 0        | 3     | 0.5    | 1    | 60.756112 |
| Puffed wheat              | Q   | C    | 50       | 2       | 0   | 0      | 1     | 10    | 0      | 50     | 0        | 3     | 0.5    | 1    | 63.005645 |
| Quaker oat squares        | Q   | C    | 100      | 4       | 1   | 135    | 2     | 14    | 6      | 110    | 25       | 3     | 1      | 0.5  | 49.511874 |
| Quaker oatmeal            | Q   | H    | 100      | 5       | 2   | 0      | 2.7   | -1    | -1     | 110    | 0        | 1     | 1      | 0.67 | 50.828392 |
| Raisin bran               | K   | C    | 120      | 3       | 1   | 210    | 5     | 14    | 12     | 240    | 25       | 2     | 1.33   | 0.75 | 39.259197 |
| name                      | mfr | type | calories | protein | fat | sodium | fiber | carbo | sugars | potass | vitamins | shelf | weight | cups | rating    |
| Raisin nut bran           | G   | C    | 100      | 3       | 2   | 140    | 2.5   | 10.5  | 8      | 140    | 25       | 3     | 1      | 0.5  | 39.7034   |
| Raisin squares            | K   | C    | 90       | 2       | 0   | 0      | 2     | 15    | 6      | 110    | 25       | 3     | 1      | 0.5  | 55.333142 |
| Rice chex                 | R   | C    | 110      | 1       | 0   | 240    | 0     | 23    | 2      | 30     | 25       | 1     | 1      | 1.13 | 41.998933 |
| Rice krispies             | K   | C    | 110      | 2       | 0   | 290    | 0     | 22    | 3      | 35     | 25       | 1     | 1      | 1    | 40.560159 |
| Shredded wheat            | N   | C    | 80       | 2       | 0   | 0      | 3     | 16    | 0      | 95     | 0        | 1     | 0.83   | 1    | 68.235885 |
| Shredded wheat 'n' bran   | N   | C    | 90       | 3       | 0   | 0      | 4     | 19    | 0      | 140    | 0        | 1     | 1      | 0.67 | 74.472949 |
| Shredded wheat spoon size | N   | C    | 90       | 3       | 0   | 0      | 3     | 20    | 0      | 120    | 0        | 1     | 1      | 0.67 | 72.801787 |
| Smacks                    | K   | C    | 110      | 2       | 1   | 70     | 1     | 9     | 15     | 40     | 25       | 2     | 1      | 0.75 | 31.230054 |
| Special k                 | K   | C    | 110      | 6       | 0   | 230    | 1     | 16    | 3      | 55     | 25       | 1     | 1      | 1    | 53.131324 |
| Strawberry fruit wheats   | N   | C    | 90       | 2       | 0   | 15     | 3     | 15    | 5      | 90     | 25       | 2     | 1      | 1    | 59.363993 |
| Total corn flakes         | G   | C    | 110      | 2       | 1   | 200    | 0     | 21    | 3      | 35     | 100      | 3     | 1      | 1    | 38.839746 |
| Total raisin bran         | G   | C    | 140      | 3       | 1   | 190    | 4     | 15    | 14     | 230    | 100      | 3     | 1.5    | 1    | 28.592785 |
| Total whole grain         | G   | C    | 100      | 3       | 1   | 200    | 3     | 16    | 3      | 110    | 100      | 3     | 1      | 1    | 46.658844 |
| Triples                   | G   | C    | 110      | 2       | 1   | 250    | 0     | 21    | 3      | 60     | 25       | 3     | 1      | 0.75 | 39.106174 |
| Trix                      | G   | C    | 110      | 1       | 1   | 140    | 0     | 13    | 12     | 25     | 25       | 2     | 1      | 1    | 27.753301 |
| Wheat chex                | R   | C    | 100      | 3       | 1   | 230    | 3     | 17    | 3      | 115    | 25       | 1     | 1      | 0.67 | 49.787445 |
| Wheaties                  | G   | C    | 100      | 3       | 1   | 200    | 3     | 17    | 3      | 110    | 25       | 1     | 1      | 1    | 51.592193 |
| Wheaties honey gold       | G   | C    | 110      | 2       | 1   | 200    | 1     | 16    | 8      | 60     | 25       | 1     | 1      | 0.75 | 36.187559 |

**Table 2** Description of the data provided in Table 1

| Sl. no. | Explanation                                                                                                                                                     |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1       | name: Name of cereal                                                                                                                                            |
| 2       | mfr: Manufacturer of cereal where A = American Home Food Products; G = General Mills; K = Kellogg's; N = Nabisco; P = Post; Q = Quaker Oats; R = Ralston Purina |
| 3       | type: cold or hot                                                                                                                                               |
| 4       | calories: calories per serving                                                                                                                                  |
| 5       | protein: grams of protein                                                                                                                                       |
| 6       | fat: grams of fat                                                                                                                                               |
| 7       | sodium: milligrams of sodium                                                                                                                                    |
| 8       | fiber: grams of dietary fiber                                                                                                                                   |
| 9       | carbo: grams of complex carbohydrates                                                                                                                           |
| 10      | sugars: grams of sugars                                                                                                                                         |
| 11      | potass: milligrams of potassium                                                                                                                                 |
| 12      | vitamins: vitamins and minerals – 0, 25, or 100, indicating the typical percentage of FDA recommended                                                           |
| 13      | shelf: display shelf (1, 2, or 3, counting from the floor)                                                                                                      |
| 14      | weight: weight in ounces of one serving                                                                                                                         |
| 15      | cups: number of cups in one serving                                                                                                                             |
| 16      | rating: a rating of the cereals                                                                                                                                 |



## Annexure C – Possible Interview Questions and Answers

1. What do you understand by “Polyglot Persistence” in NoSQL?

**Answer:** The term “Polyglot Persistence” was coined by Neal Ford in the year 2006. It refers to the use of several programming languages in one application basis the specific problems. Likewise, Polyglot persistence in NoSQL refers to the use of more than one NoSQL in an application. For example, an e-commerce application uses a graph database to make recommendations while a key-value store to store session information and shopping cart data.

2. What is CAP theorem? How is it applicable to NoSQL systems?

**Answer:** CAP theorem was given by Eric Brewer. It stands for Consistency, Availability, and Partition Tolerance. Any NoSQL obeys only two of the three characteristics (it is either CA (Consistent and Available), CP (Consistent and Partition Tolerant), or AP (Available and Partition Tolerant)).

3. Does MongoDB support primary-key, foreign-key relationship?

**Answer:** No. By Default, MongoDB does not support primary-key, foreign-key relationship.

4. Does Cassandra support ACID transactions?

**Answer:** Cassandra does not support ACID transactions. ACID transactions are supported by Relational Database Management Systems (RDBMS).

**5.** What is the difference between Replication and Sharding?

**Answer:** Replication takes the same data and copies it over multiple nodes, while Sharding puts different data on different nodes. Sharding is particularly valuable for performance because it can improve both read and write performance. Sharding helps with horizontal scaling.

**6.** Is PostgreSQL a NoSQL database?

**Answer:** No. PostgreSQL is not a NoSQL database. Examples of NoSQL databases are Cassandra, Redis, HBase, etc.

**7.** Is Neo4j a NoSQL-Document database?

**Answer:** No. Neo4j is not a NoSQL-Document type of database. Neo4j is a NoSQL-Graph type of database.

**8.** What is Apache HBase modeled after?

**Answer:** Apache HBase is a non-relational NoSQL database modeled after Google's Bigtable.

**9.** How many columns can an HBase table hold?

**Answer:** There is no limit to the number of columns an HBase table can hold.

**10.** What is HBase?

**Answer:** HBase is a schema-less NoSQL, and it defines only column families.

**11.** Which language is used in MongoDB?

**Answer:** MongoDB currently provides official driver support for all popular programming languages like C, C++, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go, and Erlang.

**12.** Which command inside aggregate command is used in MongoDB aggregation to filter the documents to allow only the documents that match the specified condition(s) to the next pipeline stage.

**Answer:** \$match command is used in MongoDB aggregation to filter the documents to allow only the documents that match the specified condition(s) to the next pipeline stage.

**13.** Which replica set node in MongoDB does not maintain its own data but exists only for voting purpose?

**Answer:** Arbiter replica set node in MongoDB does not maintain its own data but exists only for voting purpose.

**14.** Is JSON a type of NoSQL database?

**Answer:** JSON is not a type of NoSQL database. However, document databases such as MongoDB stores JSON documents.

**15.** Is Cassandra a wide-column store?

**Answer:** Yes. Cassandra is a wide-column store.

**16.** Do non-relational NoSQL databases require that schemas be defined before you can add data?

**Answer:** No. This statement is false.

**17.** Is it correct to say that wide-column is the simplest NoSQL database?

**Answer:** No. This would be false. Key-value is the simplest NoSQL database.

**18.** Is it correct to say that key-value stores can be used to store information about networks?

**Answer:** No. This would be false. Graph stores are used to store information about networks, such as social connections.

**19.** Is it correct to say that MongoDB provides high scalability with replica sets?

**Answer:** No. This would be false. MongoDB provides high availability with replica sets.

**20.** Is it correct to say that MongoDB scales horizontally using “Replication” for load balancing purpose?

**Answer:** No. This would be false. MongoDB scales horizontally using “Sharding” for load balancing purpose.

**21.** Can MongoDB Queries return specific fields of documents which also include user-defined JavaScript functions?

**Answer:** Yes. MongoDB Queries can return specific fields of documents which also include user-defined JavaScript functions.

**22.** What does the dynamic schema design of MongoDB support?

**Answer:** Dynamic schema design of MongoDB supports fluent polymorphism.

**23.** What is the purpose of “Projection” in MongoDB query statements?

**Answer:** A query in MongoDB may include “projection” to specify the fields to return from the matching documents.

**24.** What does MongoDB use to process collection of documents?

**Answer:** MongoDB processes collection of documents using Map-Reduce operations.

**25.** What protocol does Cassandra use to discover location and state information?

**Answer:** Cassandra uses a protocol called gossip protocol to discover location and state information.

**26.** Is it correct to say that SQL scales horizontally?

**Answer:** This is not true. SQL (RDBMS) scales vertically.

**27.** Is it correct to say that NoSQL databases can have flexible schema?

**Answer:** This is correct.

**28.** An E-Commerce website uses Cassandra to track the day to day activities of its customers. Which characteristics are taken into consideration for choosing Cassandra?

**Answer:** For choosing Cassandra, write heavy and highly scalable characteristics are taken into consideration.

**29.** Is it correct to say that Neo4j is not an ACID-complaint database?

**Answer:** No. This is not true as Neo4j is an ACID-complaint database.

**30.** Is it correct to say that two nodes can only be connected by a single relationship in Neo4j?

**Answer:** No. This is not true as two nodes can be connected by other relationships in Neo4j.

**31.** To how many nodes in Neo4j can a single relationship connect?

**Answer:** A single relationship can connect to one or two nodes in Neo4j.

**32.** Is it possible for a node to have multiple labels in Neo4j?

**Answer:** Yes. It is possible for a node to have multiple labels in Neo4j.



# Index

## A

- ACID properties (atomicity, consistency, isolation, and durability), [10](#), [15–16](#), [233](#)
- aggregation operations, [155–157](#)
- Allegrograph, [181](#)
- ALLOW FILTERING, [61–62](#)
  - alter commands, [75–778](#)
  - Amazon Dynamo, [39](#), [227](#)
  - analytics appliances, [4](#)
  - Apache Cassandra, [39](#), [236](#)
    - anti-entropy and read repair, [44](#)
    - application of, [40](#)
    - counter column, [73–74](#)
    - failure detection, [43](#)
    - fault-tolerance, [44](#)
    - features of, [41–45](#)
    - Gossip protocol, [43](#)
    - hinted handoff, [44–45](#)
    - memtable, [42](#)
    - nodes, [42–43](#)
    - partitioner, [43](#)
    - peer-to-peer master-less architecture, [42](#)
    - points to remember, [40](#)
    - read consistency, [44](#)
    - replication factor, [43](#)
    - SSTable (Sorted String) data file, [42](#)
    - tunable consistency, [44](#)
    - vs* RDBMS, [40](#)

- write consistency, 45
  - writes in, 44
- Apache HBase, 236
- array query operators, 154
- arrays, 142–146
  - update on, 146–153
- array update modifiers, 154
- array update operators, 154
- ASC keyword, 62
- AutoIndexID, 108

## B

- BASE (Basically Available Soft State Eventual Consistency) properties, 16–18, 40
- binary large object type (BLOB), 25
- BSON, 93
  - “bulkWrite” method, 115

## C

- CAP (Consistency, Availability and Partition tolerance) theorem, 11, 18–20
- capped collection, 108
- Cassandra Query Language (CQL), 45
  - built-in data types, 46
  - creating an index, 55
  - shell cqlsh, 46–62
- clustering columns, 51
- collections, Cassandra, 62–73
  - constraints in using, 63
  - limit on values, 63
  - list collection, 63
  - map collection, 63
  - practice on, 63–73
  - set collection, 63
  - use of, 63
  - when to use, 63
- column families, 31
- column-family databases, 11–12, 30–34
  - sample, 34
    - vs document databases, 33
    - vs key–value databases, 32–33
- column family store, 228–229
- column-oriented storage, 31
- constraints, creating, 197–198
- coordinator node, Cassandra, 43
- CouchBase, 7
- Couchbase, 236

CouchDB, 236  
COUNT() function, 195  
CREATE INDEX statement, 54–56  
CRUD operations  
    create database, 100–101  
    drop database, 101  
CSV (Comma Separated Values), 168  
    export to, 78–79  
    import from, 79–80  
cursors, in MongoDB, 161–165  
Customer Relationship Management (CRM) platforms, 1

## D

database management system, migration of, 6  
data sources, 1  
data warehousing appliances, 4  
db.dropDatabase() command, 101  
“deleteOne” statement, 116  
Delete statement, 59  
DESC keyword, 62  
DML (data manipulation statements), 53  
document database/document-oriented databases, 11, 27–30  
    differences between relational databases, 28–30  
    embedding documents or lists of values, 30  
document store, 229–231  
    limitations, 231  
DynamoDB, 236

## E

e-commerce, 222  
e-commerce application, 3  
element query operators, 154  
etworkTopologyStrategy, 43  
Evans, Eric, 4

## F

Facebook, 39  
failover, 6–7  
field update operators, 154  
filtering, 61  
“findAndModify()” method, 172

## G

getnextseq function, 172  
Google’s BigTable, 39  
Gossip protocol, 43  
graph database, 12–13, 34–36, 179–181  
    node, 36  
    popularity of, 36  
    rationale for, 181  
    relationship, 36  
    resource table, 180  
    role resource mapping table, 180  
    role table, 180  
    user role mapping table, 180  
    users table, 180  
graph data store, 3  
graph store, 231–236  
    application areas of, 233–235  
    limitations of, 233  
    pros and cons, 235–236  
    rationale for, 232–233

## H

hasNext() method, 164  
hinted handoff, 44–45  
horizontal scalability, 6

## I

index, creating, 196–197  
InfiniteGraph, 237  
“insertOne” statement, 116  
“INSERT” statement, 53–54  
Internet of Things (IoT), 2

## J

JavaScript programming, 159–161  
JSON (Java Script Object notation), 91–93, 168

## K

keyspaces, 47–52  
key-value database, 3, 12, 24–26  
    differences between relational databases, 26  
    keys, 24–25  
    namespace/keyspace, 25  
    naming convention in, 26

sample, 26  
for unrelated data, 26  
values, 25–26

key-value store  
examples, 227  
limitations, 227  
pros, 226–227  
use cases, 227

## L

list collection, 63  
logical operators to use in queries, 153

## M

map collection, 63  
map function, 158–159  
market basket analysis, 233  
MarkLogic Sever, 236  
master–slave architecture, 39, 42  
Memcached, 227, 236  
memtable, 42  
MongoDB, 7, 10, 236  
    adding new field to existing document, 119  
    aggregation operations, 155–157  
    arrays, 142–146  
    auto sharding, 91  
    collection, 93, 97  
    component set and binaries in, 99  
    cursors in, 161–165  
    database, 93  
    data types in, 101–102  
    differences between RDBMS and, 99–100  
    document in, 93, 96  
    document oriented, 90  
    drivers, 93  
    e-commerce application, 97–98  
    embedded document, 97–98  
    executing query, 159  
    features of, 89–91  
    finding documents based on search criteria, 120–131  
    finding number of documents in collection, 133–134  
    flexible schema, 98–99  
    generating unique key, 93  
    generation of unique numbers for “\_id” field, 172–173  
    indexes in, 165–168

indexing support, 91  
JavaScript programming, 159–161  
map-reduce operations, 158–161  
MongoExport, 171–172  
MongoImport, 168–171  
null values in existing documents, 131–133  
query language, 91  
rationale for, 90  
relational operators, 124, 153–154  
removing existing field from existing document, 120  
replication, 91, 94  
retrieving documents from collection, 134–135, 141  
running, 99  
scalability, 90–91  
scale-up *vs* scale-out, 100  
sharding, 94–95  
sorting documents from collection, 135–141  
SQL/RDBMS terminology and counterpart in, 95  
statements written in, 105–106  
storage of binary data, 94  
support for dynamic queries, 94  
update on arrays, 146–153  
updation of information in-place, 95

## N

Neo4j, 181, 237  
constraints, creating, 197–198  
COUNT() function, 195  
DELETE statement, 203  
deleting all nodes, 204–205  
deleting multiple nodes, 203–204  
deleting relationship, 205–206  
DROP CONSTRAINT statement, 202–203  
DROP INDEX ON statement, 201–202  
index, creating, 196–197  
MERGE command, 206–208  
nodes, creating, 181–193  
path, creating, 195–196  
retrieving nodes, 199–201  
selecting data with match, 198–199  
WHERE clause, 193–195  
network databases, 12  
next() method, 164  
nodes, creating  
    multiple nodes, 183  
    node with label, 184

node with multiple labels, 185  
node with properties, 186  
relationship between nodes, 187–190  
relationship with label and properties, 190–193  
return a created node, 186–187  
single node, 181–182  
verification of, 182  
NoSQL databases, 4  
availability of, 9–10  
benefits of using, 5  
community support, 222  
cost of, 7  
differences between SQL and, 13–14, 221–222  
distributed, 7  
e-commerce application, 222, 224–225  
features of, 4  
flexibility of, 8–9  
history of, 4  
key-value store, 226–227  
scalability of, 222  
schema, 221  
solutions using, 223  
SQL database *vs*, 224  
structured, semi-structured and unstructured data, 10–11  
types, 11–13  
understanding of, 5

## O

open-source distributed network, 4  
Oplog, 94  
OrientDB, 181, 237  
Oskarsson, Johan, 4

## P

partitioner, 43  
partition key, 50–51  
path, creating, 195–196  
polyglot persistence, 3  
    implementation of, 4  
pre-defined schema, 221  
primary key, 50–51, 58  
“project\_details” table, 61  
Project Voldemort, 236

# Q

## queries

- logical operators to use in, 153
- relational operators to use in, 153
- querying system tables, 83–84

# R

- RDBMS (relational database management system), 26
- RDBMS (SQL), 221
- read consistency of Cassandra, 44
- read repair in Cassandra, 44
- real-time analytics, 2–3
- Redis, 227, 236
- reduce function, 159
- relational databases
  - challenges faced by, 223
  - document databases *vs*, 28–30
  - finding documents based on search criteria, 122–131
  - key-value database *vs*, 26
  - null values in existing documents, 131–132
  - retrieving documents from collection, 134–135, 141
  - sorting documents from collection, 135–141
    - statements written in, 105–106
- relational operators, 124
  - array query operators, 154
  - array update modifiers, 154
  - array update operators, 154
  - element query operators, 154
  - field update operators, 154
  - logical operators to use in queries, 153
    - to use in queries, 153
- replication, 91
- replication factor, Cassandra, 43–44
- replication node, Cassandra, 43
- Riak, 227, 236
- row-oriented storage, 31

# S

- scalability, 5
- scalable system, 90–91
- SELECT statement, 83
- Select statement, 61–62
- set collection, 63
- sharding, 91, 94–95
- SimpleStrategy, 43

SQL database, 13–14  
SSTable (Sorted String) data file, 42  
standard input device (STDIN), import from, 80–82  
standard output device (STDOUT), export to, 82  
Strozzi, Carlo, 4

## T

time-to-live (TTL), 75  
TSV (Tab Separated Values) files, 168  
tunable consistency of Cassandra, 44

## U

UPDATE command, 58  
UPDATE statement, 58  
“usercounters” collection, 172  
user’s behavior and action, 228  
UTF-8 encoded string (Unicode Transformation format), 50

## V

vertical scalability, 6  
very large databases (VLDBs), 30

## W

WHERE clause, 61, 193–194  
    with multiple conditions, 194–195  
“WITH” statement, 52  
write consistency of Cassandra, 45  
writes in Cassandra, 44



## About the Book

NoSQL databases are non-relational, open-source, distributed, schema-less, and cluster friendly databases. They are hugely popular today owing to their ability to scale out or scale horizontally and the adeptness at dealing with a rich variety of data: structured, semi-structured, and unstructured data. They are malleable and flexible enough to accommodate sparse datasets, besides maintaining cost efficiency and availability.

*Demystifying NoSQL* is written in easy to comprehend language, with lots of practical/hands-on examples and exercises to help with self-study and deep dive learning. The book takes into consideration the pedagogical approach to developing the content. The book also carries practitioners' perspective and pragmatic best practices to leverage the best out of each NoSQL database. Unique insights help you pick the NoSQL solutions that are best for solving your specific data storage needs.

The book will serve the self-learning needs of students (undergraduate, postgraduate, business management graduates), IT developer's community, analysts, and other professionals. It can be a handy guide to database developers, data modelers, database users and to practically anybody who passionately works with data.

## Salient Features

- Exhaustive introduction to **NoSQL databases**.
- Examination of **Cassandra**.
- Exploration of **MongoDB**.
- Detailed explanation of **Neo4j**.
- Description of **NoSQL database orientation**.
- Supported by **real-life industrial application examples**.
- Various **self-assessment sections** with solutions.

Instructor Resources available at  
[www.wileyindia.com](http://www.wileyindia.com)

Scan Code to  
Access Videos



**READER LEVEL**  
Beginner to Advanced

**SHELVING CATEGORY**  
Engineering

**WILEY**

**Wiley India Pvt. Ltd.**

Customer Care +91 120 6291100  
[csupport@wiley.com](mailto:csupport@wiley.com)  
[www.wileyindia.com](http://www.wileyindia.com)  
[www.wiley.com](http://www.wiley.com)

*Follow us on*  
[facebook.com/wileyindia](https://facebook.com/wileyindia) [@](https://twitter.com/wileyindia) [linkedin.com/in/wileyindia](https://linkedin.com/in/wileyindia)

