MongoDB Developer

Lesson 0—Course Introduction




Table of Contents

Objectives

By the end of this course, you will be able to:

- Explain the functionalities of MongoDB as document database
- Identify the benefits of MongoDB
- Explain the different use cases of MongoDB
- Explain how to create and manage different types of indexes in MongoDB for query execution
- Explain how the replication and sharding features in MongoDB help in scaling read and write operations
- Explain the process of developing Java and Node JS applications using MongoDB

Course Overview

The course provides the following:

- Difference between NoSQL and RDBMS databases
- A detailed introduction of MongoDB as a document database
- Knowledge about MongoDB's role in the Big Data Ecosystem, Database Scaling, Replication, and Sharding
- Knowledge about CRUD operations in MongoDB
- The detailed process of developing Java and Node JS applications using MongoDB
- Detailed steps of installing MongoDB in different operating systems and performing various functions

Value to Professionals and Organizations

Course is beneficial for professionals who administer, manage, and analyze large and complex data, such as:

- Database Developers perform the following tasks:
  - Schema design
  - Developing Web application in Java and Node JS
  - Developing Mobile application in Java and Node JS

- Data Analysts perform the following tasks:
  - Perform CRUD operations
  - Analyze the data stored in MongoDB

- System and Enterprise Architects perform the following tasks:
  - Analyze use cases
  - Design system architecture
  - Scale read and write functions
  - Optimize query performance

Course Prerequisites

Fundamental Knowledge of Programming Language and Hadoop components is the basic course prerequisite. However, participants are expected to have a knowledge of SQL commands.

Lessons Covered

Following is the list of lessons covered in this course:

Lesson Number	Lesson Name
Lesson 1	Introduction to NoSQL Databases
Lesson 2	MongoDB A Database for the Modern Web
Lesson 3	CRUD Operations in MongoDB
Lesson 4	Indexing and Aggregation in MongoDB
Lesson 5	Replication and Sharding in MongoDB
Lesson 6	Developing Java and Node JS Application with MongoDB

This concludes ‘Course Introduction.’

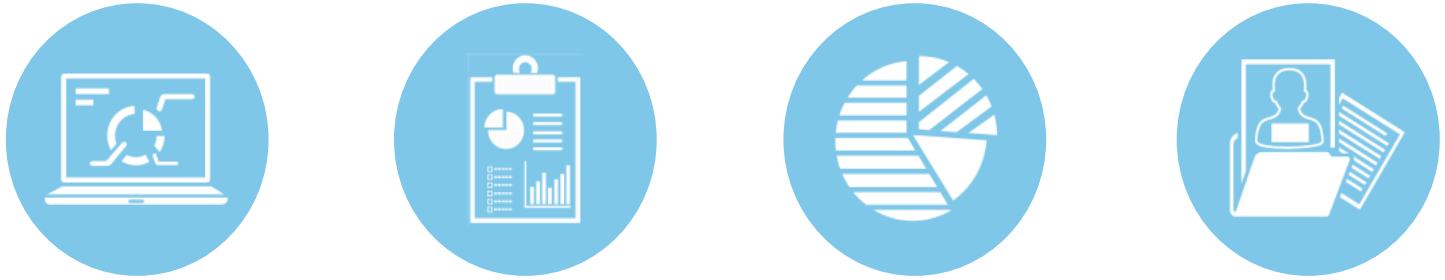
The next lesson is ‘Introduction to NoSQL Databases.’

This concludes ‘Course Introduction.’

The next lesson is ‘Introduction to NoSQL Databases.’

MongoDB Developer

Lesson1: NoSQL Database Introduction



Objectives

After completing this lesson, you will be able to:

- Explain what NoSQL databases are
- Explain the purpose of NoSQL databases
- List the benefits of NoSQL database over traditional RDBMS database
- Identify various types of NoSQL databases
- List the differences between NoSQL and RDBMS
- Explain MongoDB in relation to the CAP theorem

What is NoSQL?

Not Only SQL (NoSQL) is a new set of database that is not based on the Relational Database Management Systems (RDBMS) principles.

NoSQL was introduced by Carl Strozzi in 1998 to name his file-based database.

NoSQL represents a group of products and a various related data concepts for storage and management of large data sets.

This was the name of the hashtag (#nosql) for a meetup to discuss open source distributed databases.

What is NoSQL? (contd.)

NoSQL databases can have some common set of features such as:

- Non-relational data model
- Runs well on clusters
- Mostly open source
- Build for new generation web applications
- Is schema-less

Why NoSQL?

With the explosion of social media sites, such as Facebook and Twitter, the demand to manage large data has grown tremendously. NoSQL database:

- Resolved the challenges that were faced in storing, managing, analyzing, and archiving data
- Consists of column-based data stores, key/value pair databases, and document databases.

The NoSQL databases offers capabilities to handle large volumes of data using various available features.

Difference Between RDBMS and NoSQL Databases

Following are the differences between RDBMS and NoSQL databases:

RDBMS

- Data stored in a relational model, with rows and columns
- Follows a fixed schema
- Supports vertical scaling
- Atomicity, Consistency, Isolation, and Durability (ACID) compliant

NoSQL

- Data stored in a host of different databases—each with different data storage models
- Follows dynamic schemas
- Supports horizontal scaling
- Is not ACID complaint

Benefits of NoSQL

NoSQL solutions offer the following benefits:

- Serves as the primary datasource/operational datastore for online applications.
- Handles “big data” use cases that involve data velocity, variety, volume, and complexity.
- Offers inherent continuous availability.
- Excels at distributed database and multi-data center operations.
- Provides strong replication abilities along with read-anywhere and write-anywhere capability with full location-independence support.
- Eliminates the need for a specific caching layer to store data.

Benefits of NoSQL (contd.)

Some more NoSQL benefits include:

- Can operate in the cloud settings and exploit the benefits of cloud computing.
- Supports inclusion of additional nodes in a cluster for performance enhancement.
- Offers a flexible schema design that can be altered without downtime or service disruption.
- Supports all major operating systems, proprietary add-ons, and all common developer languages.
- Are easy to implement and use; offer sturdy functionality to handle various enterprise applications.
- Supports open source communities, which makes regular contribution to enhance the core software.

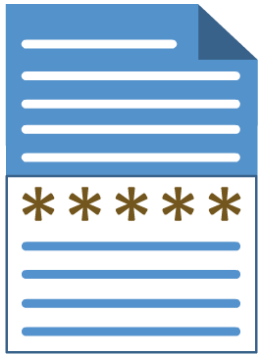
Types of NoSQL

The types of NoSQL databases are:

Key Value

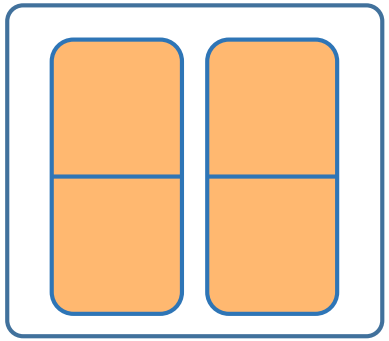
Example:
Riak, Tokyo Cabinet, Redis server, Memcached, Scalaris

Document-Based



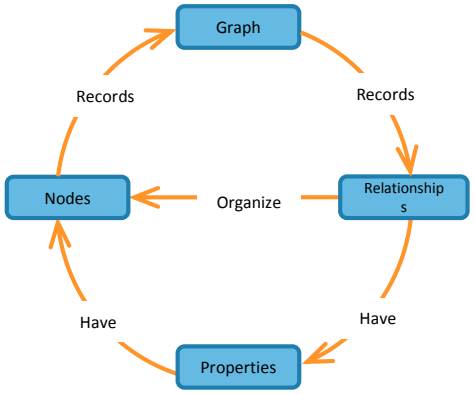
Example:
MongoDB, CouchDB, OrientDB, RavenDB

Column-Based



Example:
BigTable, Cassandra, Hbase, Hypertable

Graph-Based

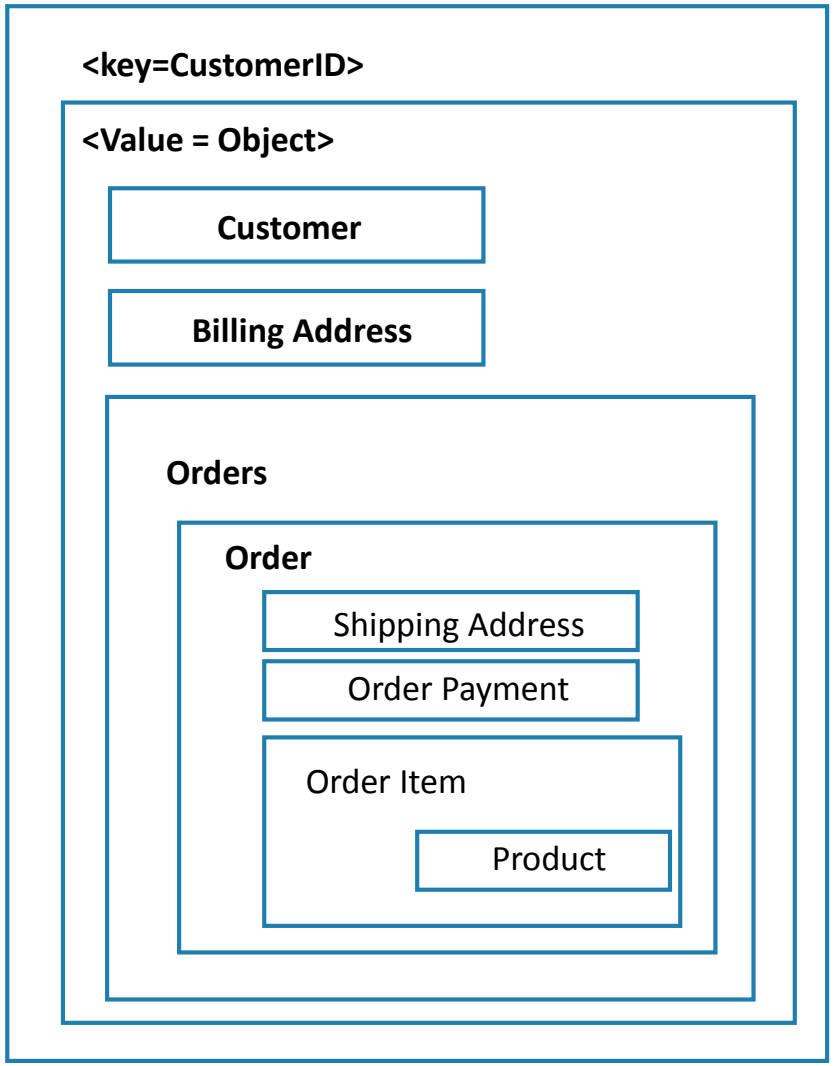


Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

Key-Value Database

Key-value stores are the simplest NoSQL databases that:

- Store every single item in the database as an attribute name (key) together with its value.
- Are ideal for handling Web scale operations that need to scale across thousands of servers and millions of users with extremely quick and optimized retrieval.



Data loss may occur in Memcached when implementing caching of user performances. The same data stored in Riak, may not be lost, but may need update.

Key-Value Database (contd.)

Following are the advantages and disadvantages of the Key-Value database.

Advantages

- Can perform queries
- Supports schemas
- Data can be accessed using a key

Disadvantages

- Does not provide any traditional database capabilities
- Maintaining unique keys are difficult as the volume of data increases

Document Database

Based on the concept of documents, the Document database:

- Stores and retrieves various documents in JavaScript Object Notation (JSON), Extensible Markup Language (XML), BSON
- Consists of maps, structures, and scalar values
- Store documents in the value part of the key-value store

Document Database Example

Examples of Document databases are:

- **MongoDB**: Provides a rich query language and many useful features such as built-in support for MapReduce-style aggregation and geospatial indexes.
- **Apache CouchDB**: Uses JSON for documents, JavaScript for MapReduce indexes, and regular HTTP for its API.

Example: (MongoDB) document

```
{Name:“SimpliLearn",
        Address:" 10685 Hazelhurst Dr, Houston, TX 77043, United States”,
        Courses: [“Big Data”,”Python”,”Android”,”PMP”,”ITIL”],
        Offices: [ “NYK”,”Dubai”,”BLR”],
        }
```

Column-Based Database

Column-based databases store data in column families as rows. These rows contain multiple columns associated with a row key.

In a Column-Based database:

- The key identifies the rows.
- The various rows need not have the same columns.

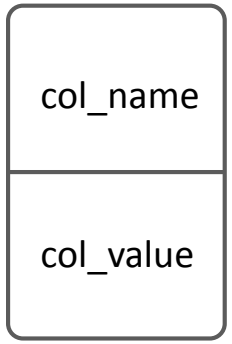
You can add a column to a row at any time without adding it to other rows.

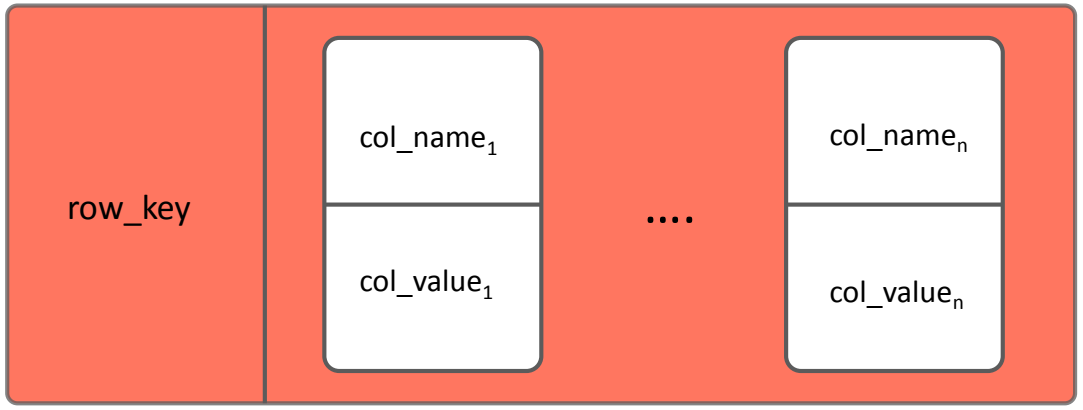
Column-Based Database (contd.)

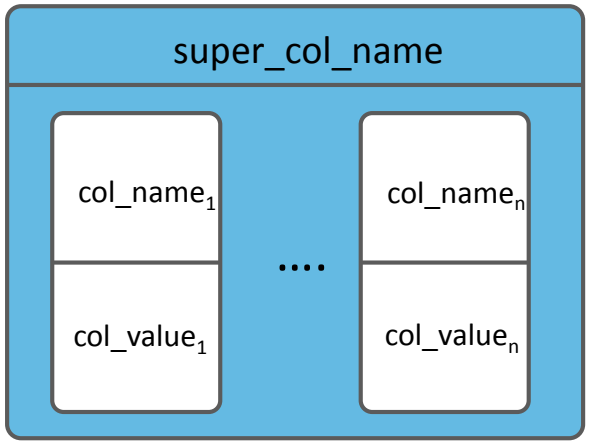
A Column-Based database reads and writes data to and from hard disk storage to quickly return a query. It lets you access individual data elements as a group in columns rather than access individually row-by-row.

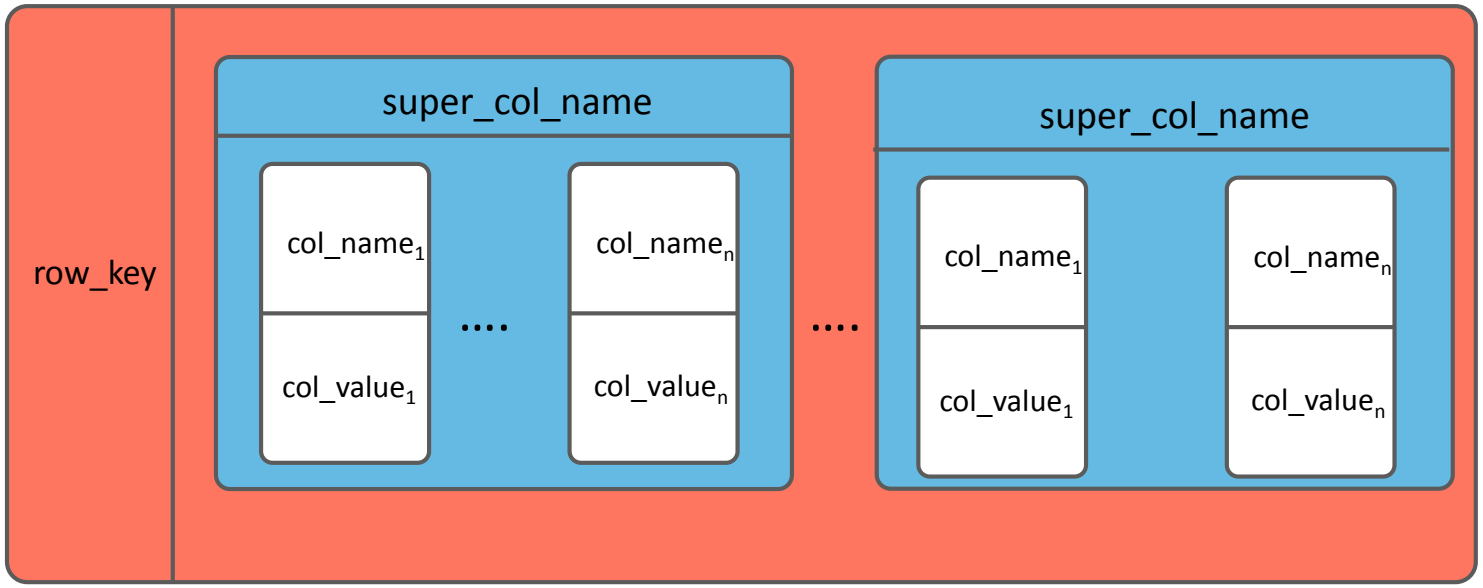
Columnar operations like MIN, MAX, SUM, COUNT, and AVG are performed quickly.

Column-Based Database (contd.)









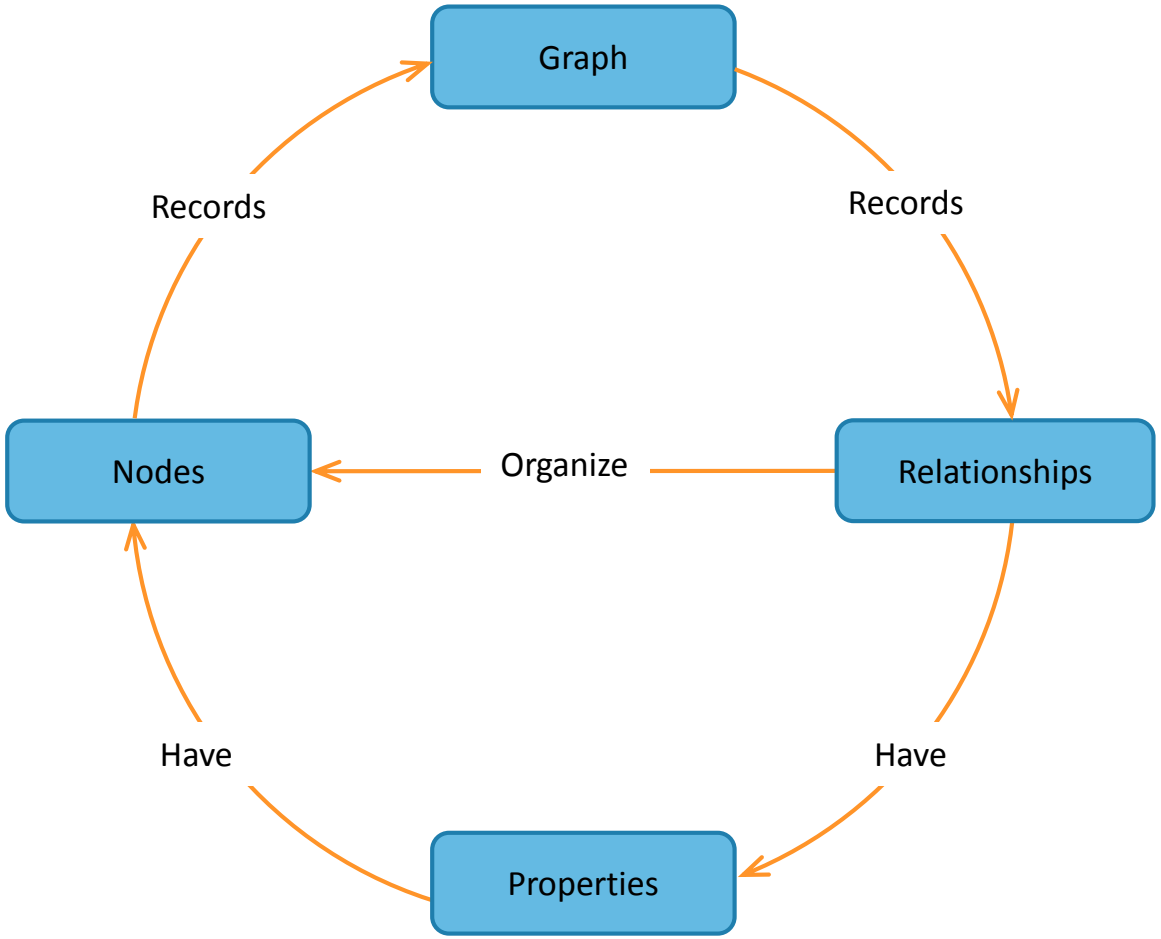
Column-Based Database Example

Cassandra is a column-based database. The features include:

- Fast and easily scalable
- Write operations spread across the cluster
- Any node can handle the read and write operations

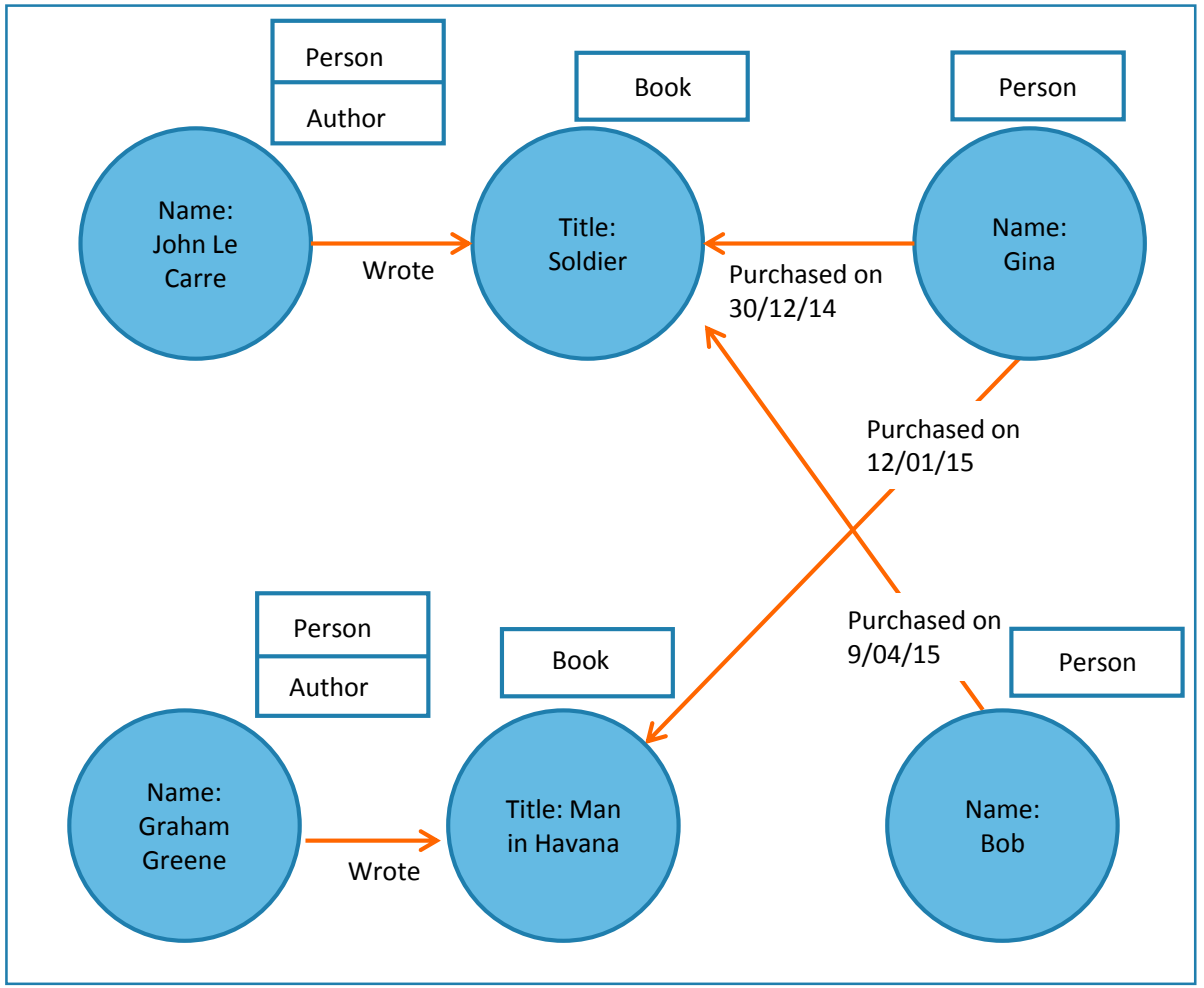
Graph Database

A Graph database makes relationships readily available for any join-like execution allowing quick access of millions of connections. It lets you store data and its relationships with other data in the form of nodes and edges.



Graph Database (contd.)

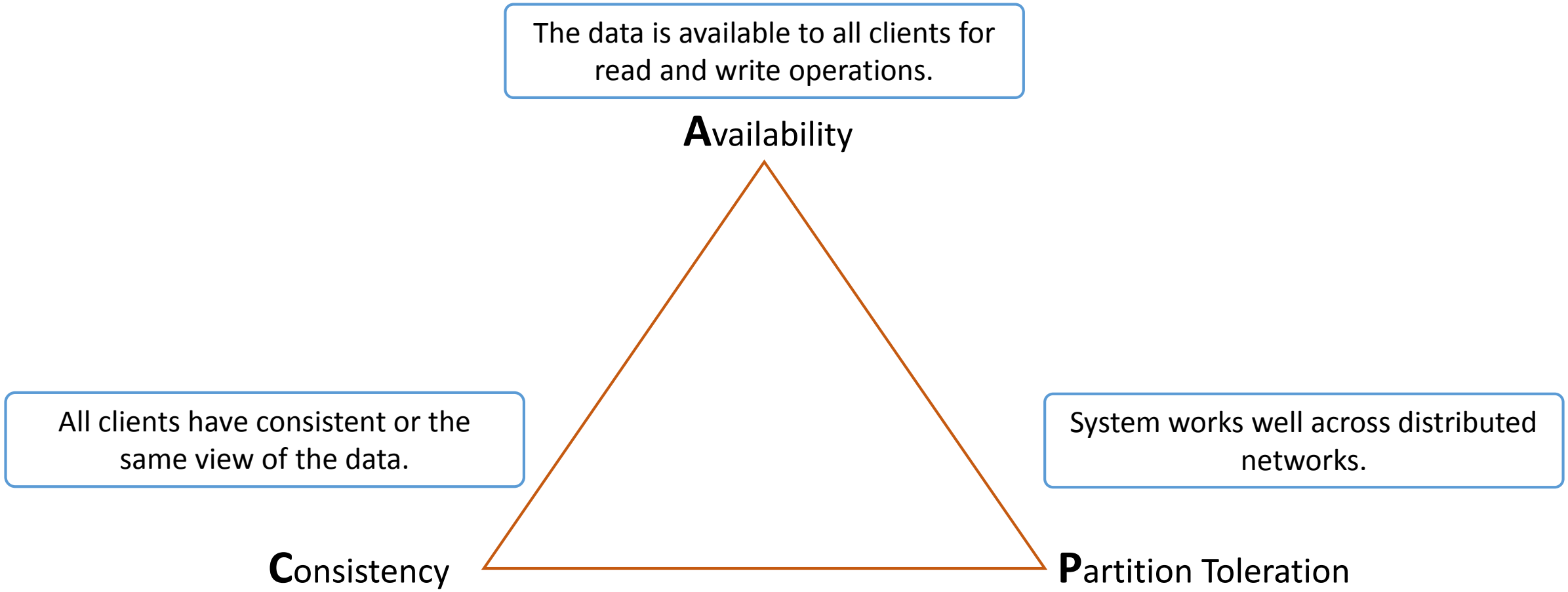
- Examples of Graph database are—Neo4J, InfiniteGraph, OrientDB, and FlockDB.
- Neo4J is ACID compliant.
- FlockDB was created by Twitter for relationship related analytics.



Labeled Property Graph Data Model

CAP Theorem

In a distributed system, the following three properties are important.



CAP Theorem (contd.)

The CAP theorem states that in any distributed system only two of the three properties can be used simultaneously—consistency, availability, or partition tolerance. To adjust the database, understanding the following requirements is essential:

- How the data is consumed by the system
- Whether the data is read or write heavy
- If there is a need to query data with random parameters
- If the system is capable of handling inconsistent data

Consistency

Below are the definitions of consistency in CAP theorem and ACID.

CAP Theorem

Consistency in CAP theorem means consistent read and write operations for the same sets of data so that concurrent operations see the same valid and consistent data state.

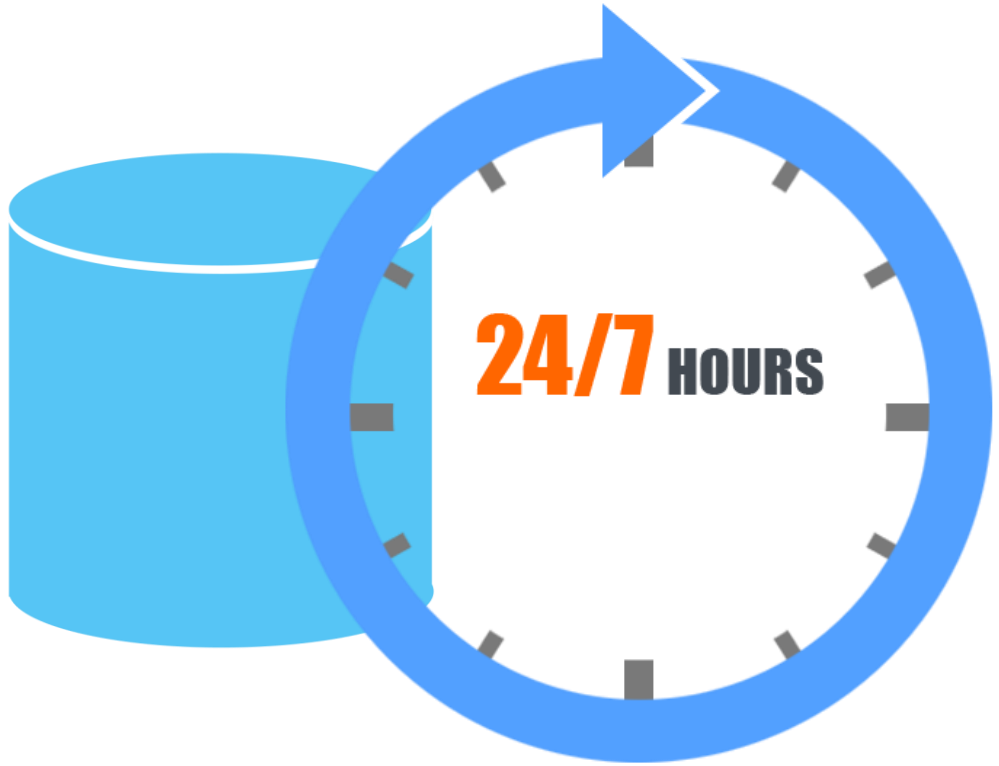
ACID

Consistency in ACID means if the data does not satisfy predefined constraints, it is not persisted.

Availability

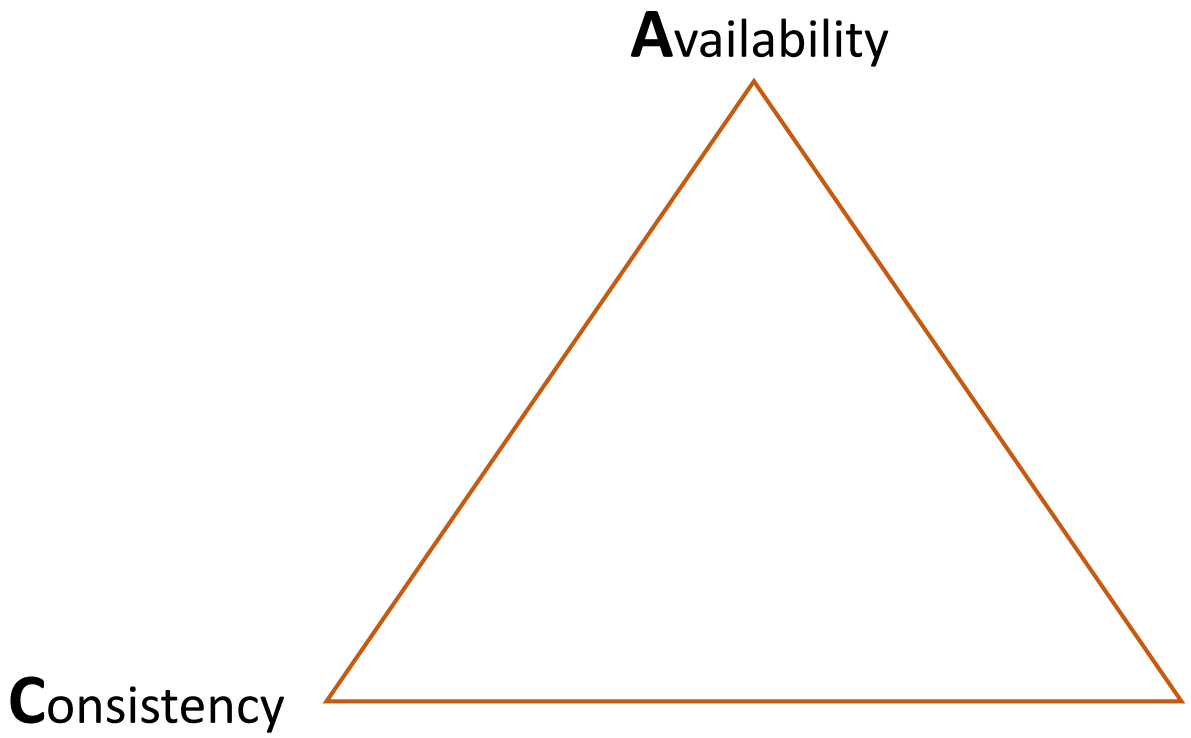
Availability means:

- The database system is available to operate when required.
- If a system is not available to serve a request when needed, it is not available.



Partition Tolerance

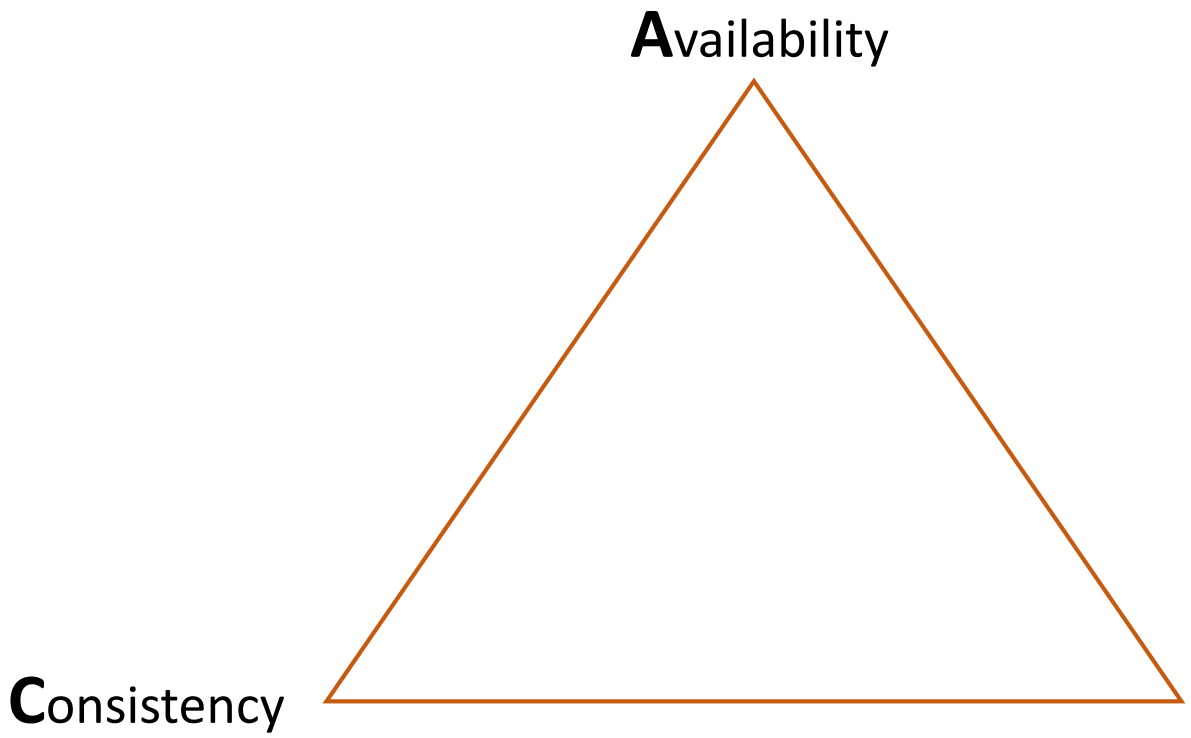
Partition tolerance or fault-tolerance is the third element of the CAP theorem. It measures the ability of a system to continue its service when some of its clusters become unavailable.



Partition tolerance measures the ability of a system to continue when some of its clusters become unavailable.

Partition Tolerance

Partition tolerance or fault-tolerance is the third element of the CAP theorem. It measures the ability of a system to continue its service when some of its clusters become unavailable.



Partition tolerance measures the ability of a system to continue when some of its clusters become unavailable.

Quiz

QUIZ 1	Which of the following is a benefit of using NoSQL database?

a. Scalability

b. Flexible schema

c. Open source technology

d. All of the above

QUIZ 1

Which of the following is a benefit of using NoSQL database?

a. Scalability

b. Flexible schema

c. Open source technology

d. All of the above

The correct answer is **d**.

Explanation: NoSQL database provides scalability, flexible schema, and open source technology.

QUIZ 2

Which of following is type of Document database ?

a. MongoDB

b. Redis server

c. Riak

d. Cassandra

QUIZ 2

Which of following is type of Document database ?

a. MongoDB

b. Redis server

c. Riak

d. Cassandra

The correct answer is **a**.

Explanation: MongoDB is document database that stores data as BSON.

QUIZ 3	According to CAP theorem, what kind of a system is MongoDB?

a. CA

b. CP

c. AP

d. CAP

QUIZ 3	According to CAP theorem, what kind of a system is MongoDB?

a. CA

b. CP

c. AP

d. CAP

The correct answer is **b**.

Explanation: MongoDB is CP system.

Summary

Here is a quick recap of what was covered in this lesson:

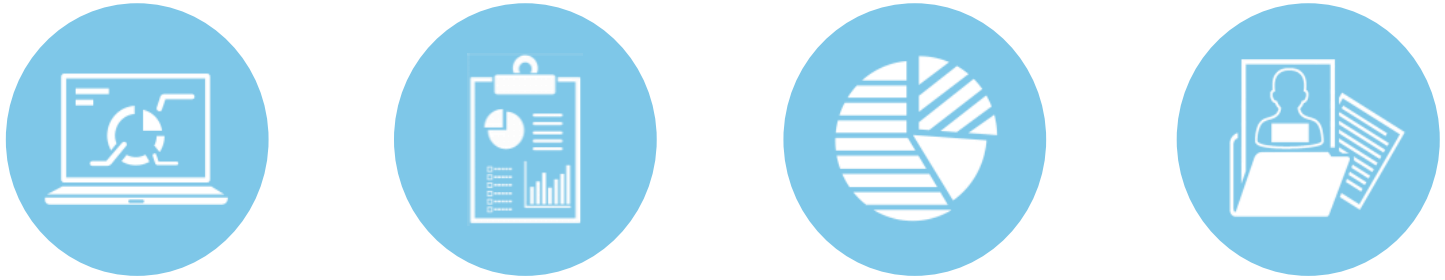
- NoSQL represents a class of products and a collection of diverse or related data concepts for storage and manipulation.
- NoSQL databases are used to efficiently manage large-volume and semi-structured data.
- The four basic NoSQL database types are— Key-Value, Document-based, Column-based, and Graph-based.
- According to the CAP theorem, a distributed computer system cannot provide all the three properties together—consistency, availability, and partition tolerance.

This concludes ‘Introduction to NoSQL Databases’.

The next lesson is ‘Database for the Modern Web’.

MongoDB Developer

Lesson 2: MongoDB–A Database for the Modern Web



After completing this lesson, you will be able to:

- Describe what MongoDB is
- Identify the key features of MongoDB
- Explain MongoDB's core server and tools
- Explain how to install MongoDB on Windows and Linux computers
- Identify the steps to start the MongoDB server
- Identify the data types available in MongoDB
- Identify the schema design and data modeling techniques in MongoDB

What is MongoDB?

MongoDB replaces the concept of ‘rows’ of traditional relational data models with ‘documents’. Following are two key characteristics of MongoDB:

Document Based

- Allows embedded documents, arrays, and represents a complex hierarchical relationship using a single record.

Schema Free

- The keys used in documents are not predefined or fixed. Without a fixed schema, massive data migrations have become unnecessary.

MongoDB offers developers the flexibility of working with evolving data models.

JSON

JavaScript Object Notation (JSON) is an open data interchange format that:

- Is language independent.
- Uses conventions of the C-family of languages, Java, JavaScript, Perl, and Python.
- Supports the basic data types, such as numbers, strings, boolean values, arrays, and hashes.

JSON Structure

JSON is built on the following two structures:

1. A collection of name/value pairs, such as an object, record, struct, dictionary, hash table, keyed list, or associative array.

2. An ordered list of values, such as an array, vector, list, or sequence.

```
{
        "_id" : 1,
        "name" : { "MongoDB"},
        "customers" : [ "Metlife", "OTTO", "Expedia", "ADP" ],
        "applications" : [
                { "name"   : "Forward",
                        "domain" : "e-commerce"
                },
                { "name"   : "Calipso",
                  "domain" : "content management"
                }
        ]
}
```

BSON

- Binary JSON or BSON is a binary serialization format which is used for storing documents and making remote procedure calls in MongoDB. BSON has the following three characteristics:

Lightweight

- When used over the network, BSON keeps the overhead involved in processing extra header data to a minimum.

Traversable

- It is designed to traverse easily across network. This helps in its role as the primary data representation for MongoDB.

Efficient

- It uses C data types that allow easy and quick encoding and decoding of data.

MongoDB Structure

The table and view structure is known as collection, which:

- Group documents that are structurally or conceptually similar.
- Embed related documents to query related data.
- Distribute data for a table among different shards similar to partition in Relational Database Management Systems (RDBMS).
- Are grouped into databases.

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (JSON, BSON)
Column	Field
Index	Index
Join	Embedded Document
Foreign Key	Reference
Partition	Shard

Data related to a single application should be stored at one database. Use separate databases when storing multiple application or users data on the same MongoDB server.

Document Store Example

```
> db.user.findOne({age:35})
{
        "_id" : ObjectId("5224e0bd52…"),
        "first" : “Mark",
        "last" : “Tailor",
        "age" : 35,
        "interests" : [
                “long distance running",
                "Mountain Biking ]
        "favorites": {
                "color": “Yellow",
                "sport": “Boxing"}
```

MongoDB as a Document Database

The data model in MongoDB is document-based, which does not conform to any pre-specified schema. The differences between a relational database and MongDB are as follows:

Relational Database	MongoDB
Rows are stored in a table and each table has a strictly defined schema that specifies the permitted types and columns. If a row in a table requires an extra field, the entire tabled needs to be altered.	Groups documents into collections that do not follow any schema. Each individual document in a collection can have a completely independent structure.

A schema less model lets you represent data with variable properties.

Transaction Management in MongoDB

Transactions in MongoDB are as follows:

- MongoDB supports single phase commit at each document level.
- A write operation in a single document is considered atomic in MongoDB even if the operation alters multiple embedded documents.

A write operation is atomic but the entire operation is not atomic and other operations may interleave.

Easy Scaling

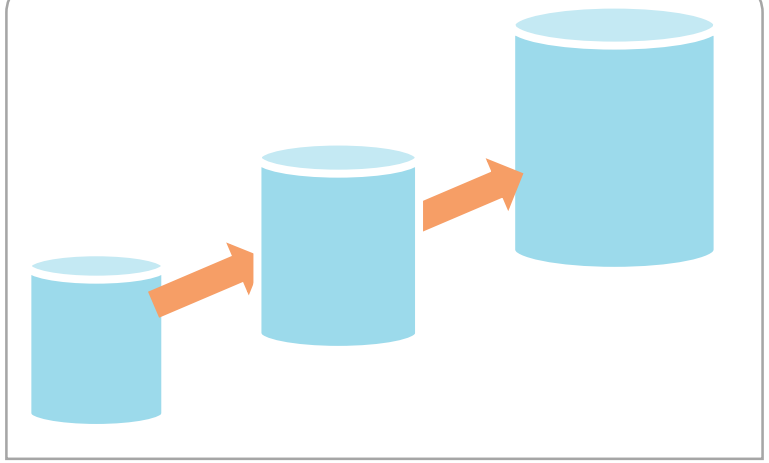
Data set sizes are growing with the growing technology. This has created a demand for:

- More data storage with more databases capabilities.
- Scaling databases for size and performance.

Scaling Up vs. Scaling Out

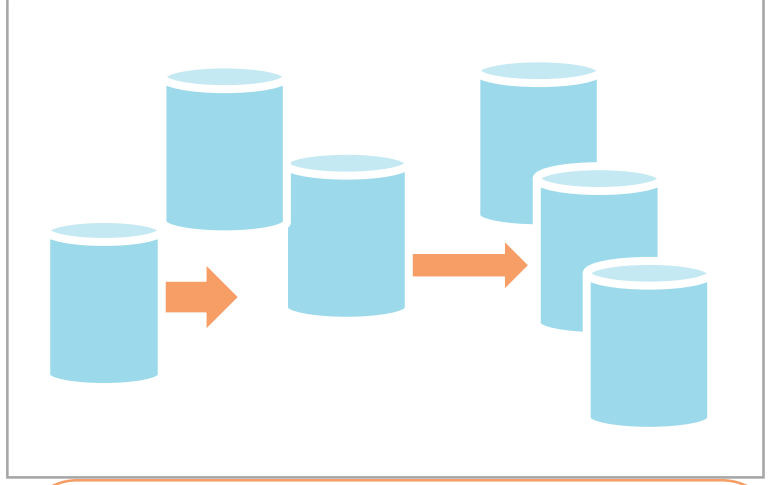
Scaling a database involves two choices:

Scaling Up



- Easy but costly affair
- Even the most powerful database may not be able to manage growing volumes of data

Scaling Out



- Extensible and cost-effective
- Commodity server can be added to increase storage and enhance performance

Vertical Scaling

A database can be scaled by upgrading the hardware. The technique of enhancing a single node hardware is called vertical scaling or scaling up. Vertical scaling is:

- Simple
- Reliable
- Cost-effective to some extent

If your database is running on a physical hardware, and the cost of a more powerful server is not permitted, consider horizontal scaling.

Horizontal Scaling

Horizontal scaling distributes the database across multiple machines. The advantages include:

- Commodity hardware are used.
- Mitigates the risk of failure.
- Failure is less disastrous because a single machine is only a small part of the entire system.

MongoDB supports horizontal scaling and uses a range-based partitioning mechanism called 'auto-sharding' which:

- Automatically manages data distribution across multiple nodes.
- Allows addition of new nodes.
- Is transparent to the client.

Features of MongoDB

Following are some key features of MongoDB:

Ad hoc queries	Performs search functions by field, range queries, and regular expression searches.
Querying	Uses rich query language and complex conditions to retrieve documents.
Fast In-Place Updates	Allows write semantics, enable journaling, and control the speed and durability.
Server-side JavaScript execution	Uses JavaScript in queries and aggregation functions such as MapReduce, which are sent to the database for execution.
Capped Collections	Maintains insertion order and once the specified size has been reached, behaves like a circular queue.

Secondary Indexes

MongoDB implements multiple secondary indexes as B-trees that can be optimized for range scan queries and queries with sort clauses.

MongoDB allows creation of up to 64 indexes per collection and supports indexes, such as ascending, descending, unique, compound-key, and geospatial.

MongoDB uses the same data structure for indexes as most RDBMSs.

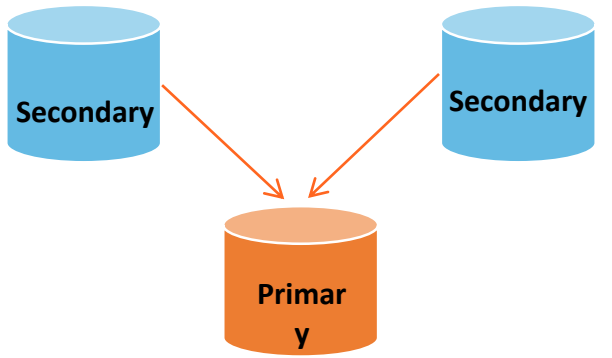
Replication

MongoDB uses replica sets to create database replication. A replica set performs the following actions:

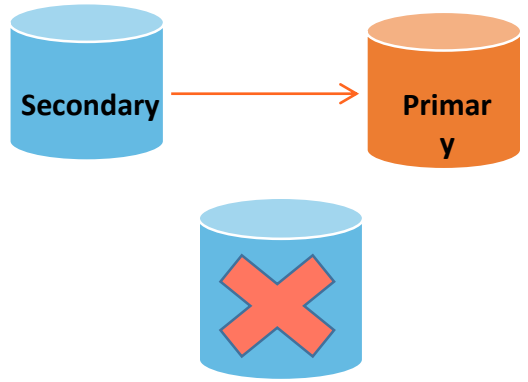
- Distributes data across various MongoDB nodes called shards for redundancy.
- Automates failover when server or network outages occur.
- Scales database reads.

A replica set contains one primary node and one or more secondary nodes, the primary node supports both read and write, whereas the secondary supports read.

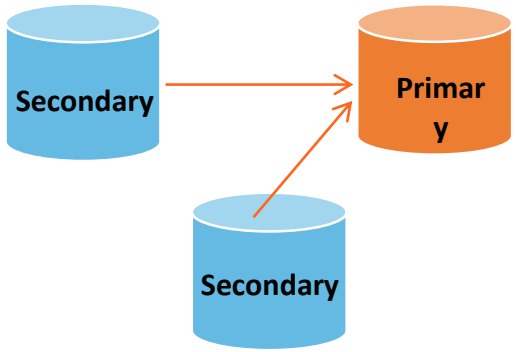
Replication (contd.)



A Working Replica Set



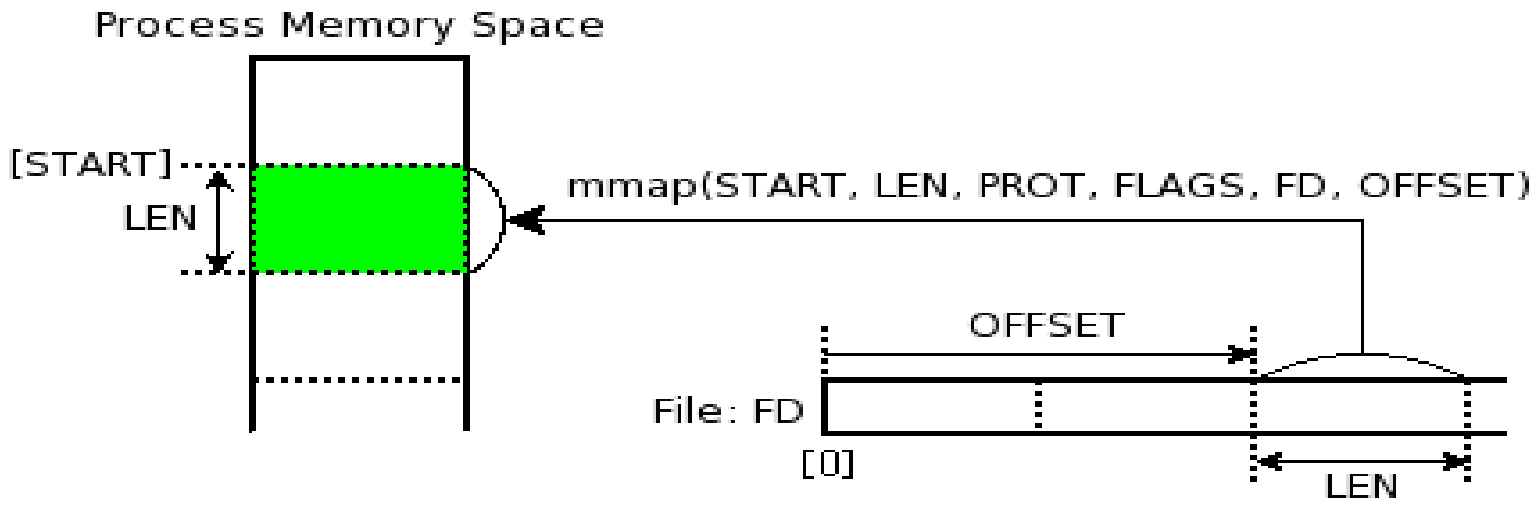
Secondary node is converted to primary when the primary fails



Original primary comes back as secondary

Memory Management

MongoDB stores the data in memory mapped files and uses all the system memory for these mapped files for faster operations. MongoDB allows its operating system to manage its memory which impacts its performances and operations.



Replica Set

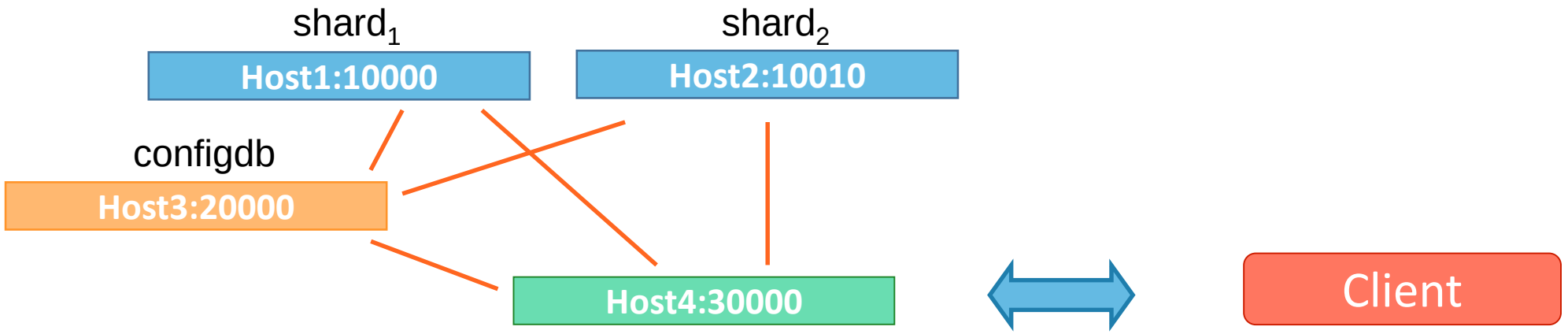
The replica set feature in MongoDB facilitates redundancy and failover with the following actions:

- When a master node of the replica set fails, another member of replica set is converted to the master.
- Allows choosing a master or slave depending on whether you want a strong or delayed consistency.
- Keeps replicated data on the nodes belonging to different data centres to protect the data in the case of natural disaster.

Auto Sharding

MongoDB uses sharding for horizontal scaling. For automatic sharding, choose a shard key that determines the distribution of the collection data.

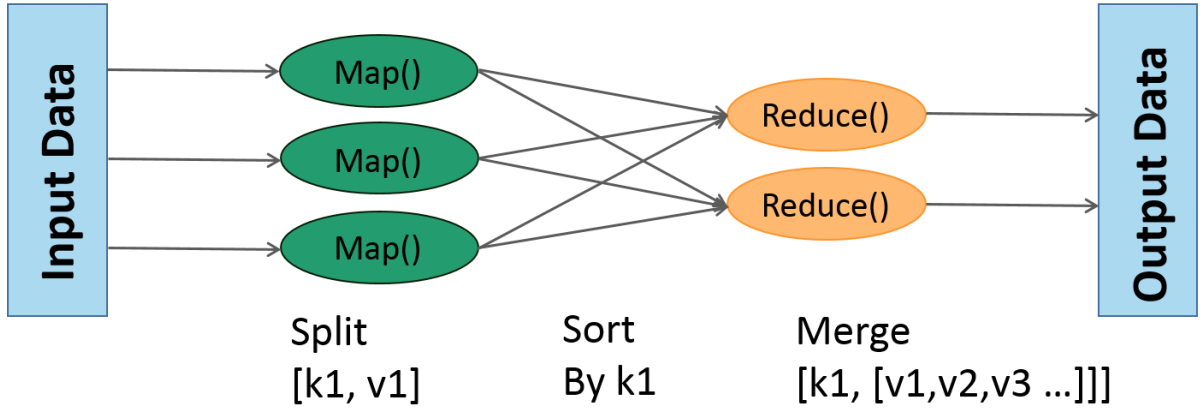
MongoDB is spread over multiple servers and performs load balancing and/or data duplicating to keep the system up and running in case of hardware failure.



Aggregation operations analyze data sets and return calculated results.

A mapreduce operation consists of two phases:

- **Map**: Documents are processed and one or more objects are produced for each input document.
- **Reduce**: The outputs of the map operation are combined.



Map reduce can define a query condition to select the input documents, sort and limit the results.

Collection and Database

A collection is a group of documents not restricted by any schema. MongoDB groups collections into databases. A single instance of MongoDB can host several independent databases.

Having different types of documents in the same collection can be cumbersome for developers and administrators. Developers need to ensure that their queries retrieve specific documents or their application codes handle documents of different structure.

Storing data of a single application in the same database is a good practice.

Schema Design and Modeling

The collections in MongoDB allows flexibility in document structure that enables easy mapping of documents to an entity or an object. All documents in a collection share a similar structure.

The key challenge in data modeling involves maintaining a balance between:

- The needs of the application
- The performance capability of the database engine
- Data retrieval patterns

When designing a data model consider the application usage of the data, such as queries, updates, and data processing along with its inherent structure.

Reference Data Model

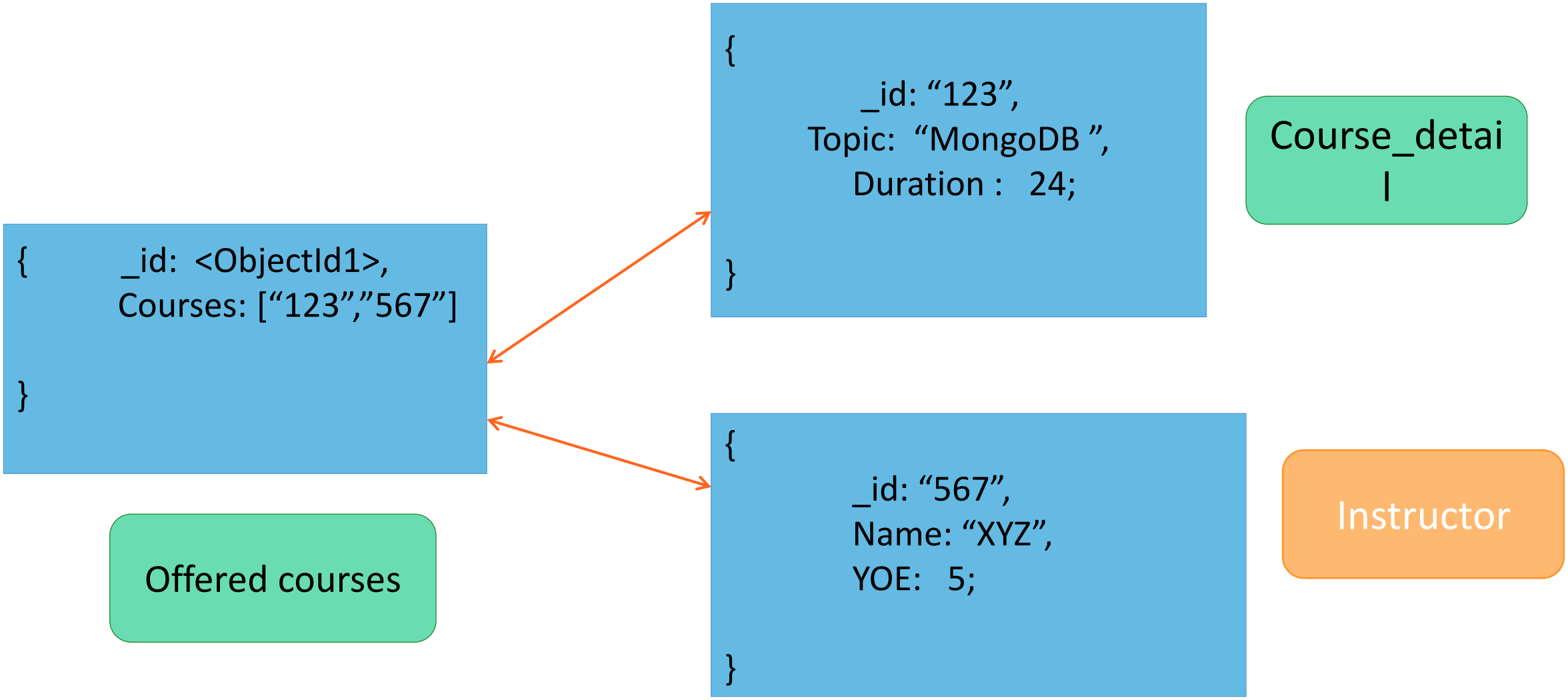
Data model design requires two important things—the structure of documents and representation of the data relationships. References and embedded documents allow applications to represent data relationship.

References use links to store data relationships and applications use these references to access the related data. These are normalized data models, which you can use:

- When embedding document results in data duplication does not give sufficient performance benefit.
- To represent complex many-to-many relationships.
- To model large hierarchical data sets.

Normalized data models can send more database queries from client to the database server.

Reference Data Model Example



Embedded Data Model

Embedded documents store related data in a single document structure and thus capture relationships between data. MongoDB allows denormalized data models to permit retrieval and manipulation of related data in a single database operation.

Embedded data models can be used when the following relationships exist between entities:

- ‘contains’
- one-to-many

Embedded data models allow related data update in a single atomic write operation.

Embedded Data Model Example

```
{
        _id: “123”,
          coursedetail:
        {
                Topic:  “MongoDB ”,
                Duration : 24,


        }

        instructor:
          {
                name: “XYZ”,
                YOE: 5,
        }

}
```

Data Types

MongoDB supports the following data types:

- **Null:** Used to represent both a null value and a nonexistent field
  {"x”: null}
- **Undefined:** Used in documents
  {"x" : undefined}
- **Boolean:** Used for the values 'true' and 'false'
  {"x" : true}
- **32-bit integer:** Cannot be represented on the shell
- **64-bit integer:** The shell cannot represent these
- **64-bit floating point number:** All numbers in the shell will be of this type

Data Types (contd.)

Some more data types supported by MongoDB are:

- **Maximum value:** Contains a special data type that represents the largest possible value. The shell does not support this type.
- **Minimum value:** Contains a special data type that represents the smallest possible value. The shell does not support this type.
- **ObjectId:** Unique, fast to generate and ordered, these consists of 12 bytes where the first four bytes represent the ObjectId creation time.
- **String**: Are UTF-8 compliant. When serializing and deserializing BSON, programming languages convert language strings to UTF-8 format.
- **Symbol**: Not supported by the shell. When the shell gets a symbol, it converts it into a string.
- **Timestamps:** BSON offers a special timestamp for internal MongoDB use that is not associated with the regular Date type.
- **Date:** A 64-bit integer that denotes the number of milliseconds since the UNIX epoch.
- **Regular expression:** Documents contain JavaScript's regular expression syntax.

  {"x" : /simplilearn/i}

Data Type (contd.)

Some more data types are:

- **Code:** Documents can contain JavaScript code. For example:

  {"x" : function() { /* ... */ }}

- **Binary data:** A string of arbitrary bytes that cannot be manipulated from the shell.
- **Array:** Sets or lists of values can be represented as arrays. For example:

  {"courses" : ["PMP", "Cloud", " MongoDB "]}

- **Embedded document:** Documents can contain entire documents, embedded as values in a parent document. For example:

  {"course_duration" : {"MongoDB " : "24 Hrs"}}

Core Servers of MongoDB

The core database server of MongoDB can be run on executable process called mongod or mongodb.exe on Windows.

A mongod process can be run on several modes:

- Replica set: Configurations comprise two replicas and an arbiter process that reside on a third server.
- Per-shard replica sets: The auto-sharding architecture of MongoDB consist of mongod processes configured as per-shard replica sets.
- Mongos: A separate routing server is used to send requests to the appropriate shard.

Mongos queries from the application layer and locates the data in the sharded cluster to complete its operations.

MongoDB's Tools

MongoDB tools consists of the following:

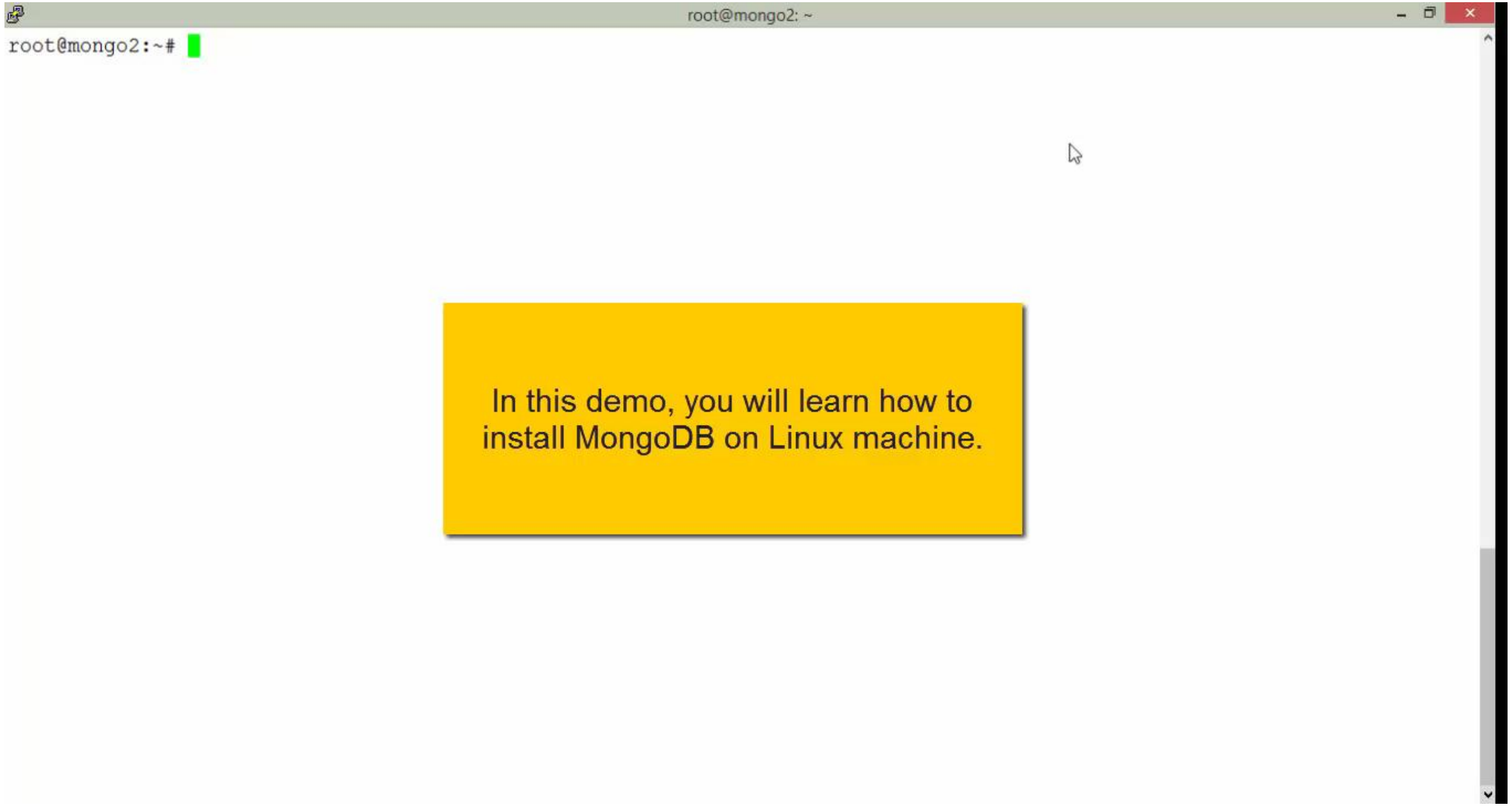
- **The JavaScript shell:** The MongoDB command shell is JavaScript-based used for administering the database and manipulating data.
- **Database drivers :** Provides an Application Program Interface (API) that matches the syntax of the language used.
- **Command-line tools:**
  - mongodump and mongorestore: Standard utilities that helps backup and restore a database.
  - mongoexport and mongoimport: Used to export and import JSON, comma separated value (CSV), and Tab separated Value (TSV) data.
  - Mongosniff: A wire-sniffing tool used for viewing operations sent to the database.
  - Mongostat: Provides helpful statistics, such as inserts, queries, updates, deletes, and so on.

Installing MongoDB on Linux

This demo will show the steps to install MongoDB on Linux 64 bit.

Please refer to e-learning videos for this demo.

Installing MongoDB on Linux



Installing MongoDB on Windows

This demo will show the steps to install MongoDB on Windows. Please refer to e-learning videos for this demo.

Installing MongoDB on Windows

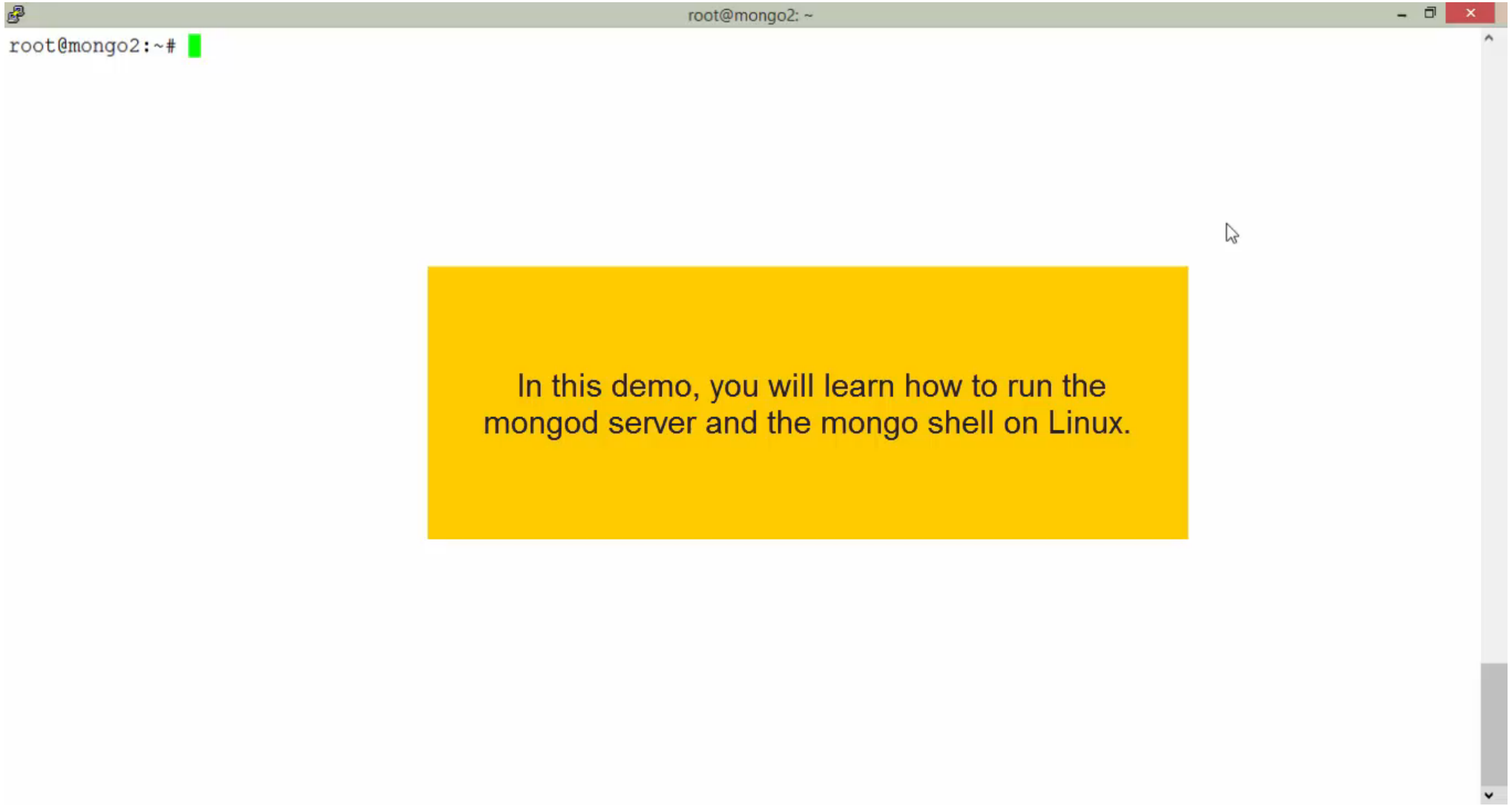
This demo will show the steps to install MongoDB on Windows. Please refer to e-learning videos for this demo.

Starting MongoDB On Linux

This demo will show the steps to start MongoDB on Linux.

Please refer to e-learning videos for this demo.

Starting MongoDB On Linux

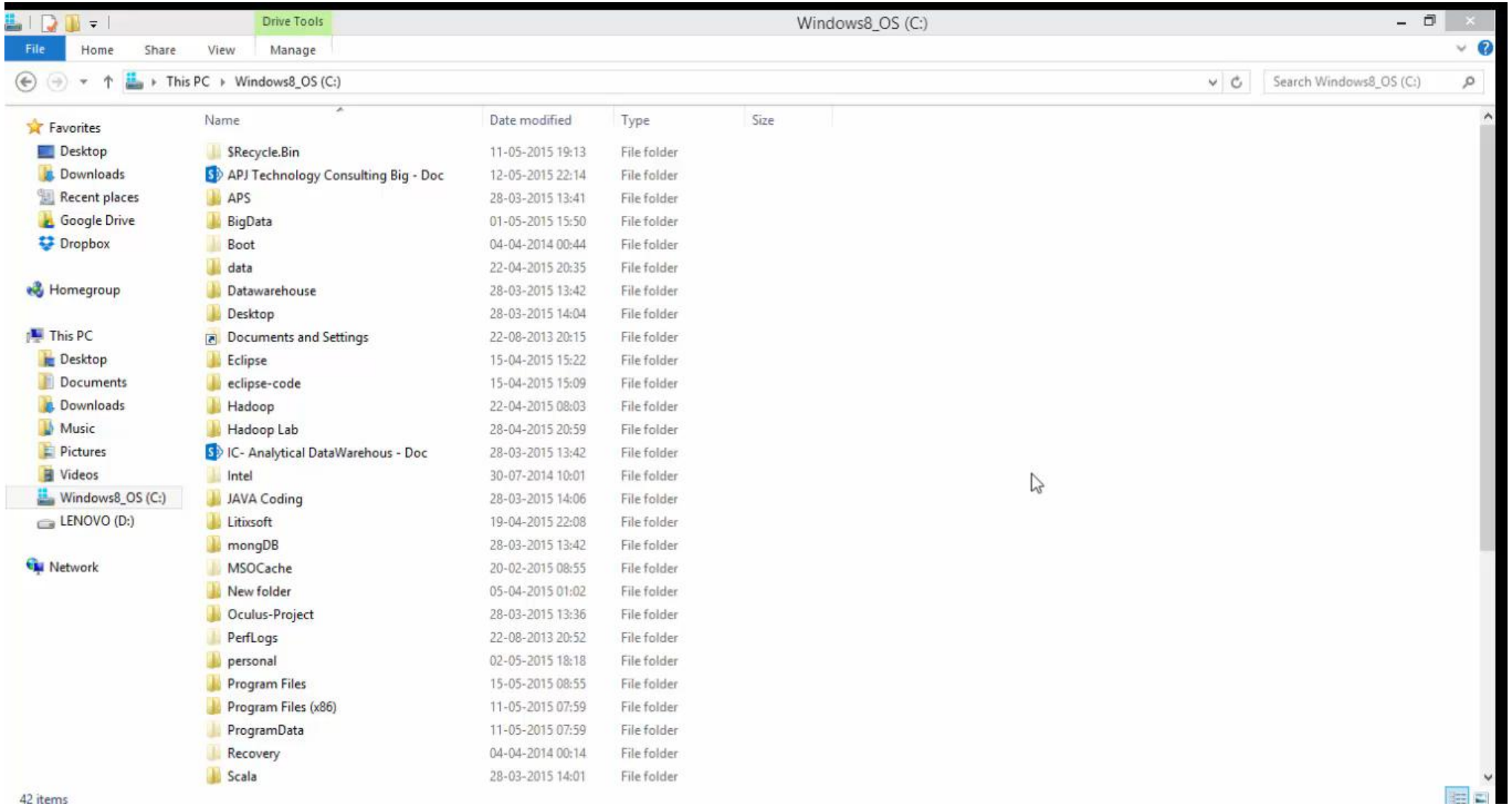


Starting MongoDB On Windows

This demo will show the steps to start MongoDB on Windows.

Please refer to e-learning videos for this demo.

Starting MongoDB On Windows



- Some use cases of MongoDB are:

Personalization

- Allows personalization of customer experience in real time to predict the wants and needs of customers.

Mobile

- Allows scaling of mobile applications to cater to millions of online users.

Internet of Things (IOT)

- Manages huge volumes of data generated by IOT. Allows building of your own IOT suite to manage big data on your own.

Real-time Analytics

- Allows real-time data analysis, minute by minute and second by second.

Use Cases (contd.)

- Some more use cases of MongoDB are:

Web Application

- Helps Web application manage data efficiently through rich data structures, such as documents.

Content Management

- Allows storing and presenting of any type of content, build new features, incorporate all kinds of data in a single database.

Catalog

- Allows access to customer data to provide online shoppers a highly personalized and smooth shopping experience.

Single View

- Allows building of a single view of all your business data.

Quiz

QUIZ 1	In which of the following format does MongoDB store data?

a. JSON

b. BSON

c. XML

d. HTML

QUIZ 1	In which of the following format does MongoDB store data?

a. JSON

b. BSON

c. XML

d. HTML

The correct answer is **b**.

Explanation: MongoDB stores the data into BSON format.

QUIZ 2

Which among the following is the core server of MongoDB ?

a. mongos

b. Config server

c. Routing server

d. mongod

QUIZ 2

Which among the following is the core server of MongoDB ?

a. mongos

b. Config server

c. Routing server

d. mongod

The correct answer is **d**.

Explanation: mongod is the core server of the MongoDB.

QUIZ 3

Which among following is analog of an RDBMS row in MongoDB?

a. Database

b. Schema

c. Collection

d. Index

QUIZ 3

Which among following is analog of an RDBMS row in MongoDB?

a. Database

b. Schema

c. Collection

d. Index

The correct answer is **c**.

Explanation: In MongoDB, collection is analog of an RDBMS row.

Summary

Here is a quick recap of what was covered in this lesson:

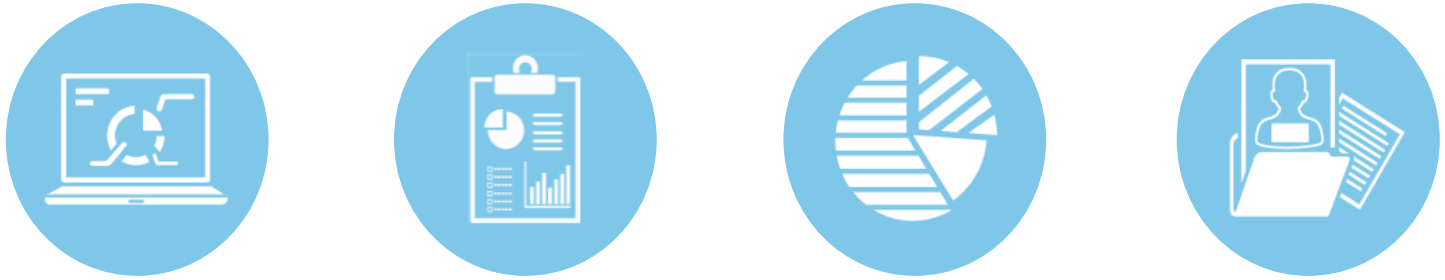
- MongoDB is a document-based database that represents a complex hierarchical data relationship using embedded document model or using reference model.
- MongoDB uses JSON as the data format, which is based on:
  - A collection of name/value pairs
  - An ordered list of values
- A collection in MongoDB is a table and view structure that groups structurally or conceptually similar documents.
- MongoDB supports auto-sharding to manage data distribution across multiple nodes.
- MongoDB uses replica sets to create redundancy and automates failover when server or network outages occur.

This concludes ‘MongoDB: A Database for the Modern Web’.

The next lesson is ‘CRUD Operations in MongoDB’.

MongoDB Developer

Lesson 3: CRUD Operations in MongoDB



After completing this lesson, you will be able to:

- Explain how to perform data modification in MongoDB
- Explain how to perform a batch insert, ordered bulk insert, and unordered bulk insert
- Explain how to insert documents to MongoDB collection
- Explain how to modify, update, upsert and delete documents from MongoDB collection
- Identify the steps to use different query modifiers such as find, Query Conditionals, Queries, $not, Regular Expressions, and so on
- Identify the steps to retrieve documents by using the find query, Or condition, cursor, batch insert
- Identify the steps to retrieve documents for array fields
- Identify the steps to perform a batch insert, ordered bulk insert, and unordered bulk insert

Data Modification in MongoDB

Data modifications in MongoDB involve creating, updating, or deleting data. These operations modify the data of a single collection. The Insert() method is used to insert a document into a MongoDB collection.

An example of the Insert query is given below.

```
db.courses.insert({Name:“SimpliLearn",
Address:" 10685 Hazelhurst Dr, Houston, TX 77043, United States”,
Courses: [“Big Data”,”Python”,”Android”,”PMP”,”ITIL”],
Offices: [ “NYK”,”Dubai”,”BLR”]
 });
```

Batch Insert in MongoDB

A batch insert allows storing of multiple documents in a database at one time. Following are the characteristics of a batch insert:

- Sends hundreds or thousands of documents to the database in a batch at one time
- Inserts are faster
- Reduces insert time by eliminating many header processing activity
- Batch inserts are used in applications for storing server logs and sensors data
- Limits inserts to 16MB in a single batch insert

Ordered Bulk Insert

MongoDB groups ordered list operations by its type and contiguity. Characteristics of ordered bulk insert include the following:

- Executes the write operations serially
- If an error occurs during one write operation, MongoDB returns the remaining write operations
- Each group of operations can have maximum 1000 operations
- On exceeding 1000 operations, MongoDB divides the group into smaller groups of 1000 or less

```
var bulk = db.items.initializeOrderedBulkOp();
        bulk.insert( { _id: 1, item: “pen", available: true, soldQty:700} );
        bulk.insert( { _id: 2, item: “pencil", available:false, soldQty:900} );
        bulk.insert( { _id: 3, item: “books", available: true, soldQty: 600 } );
        bulk.execute();
```

Performing Ordered Bulk Insert

This demo will show the steps to perform an ordered bulk insert in MongoDB. Please refer to e-learning videos for this demo.

Unordered Bulk Insert

In an unordered operations list, MongoDB can execute in parallel in a nondeterministic manner.

When performing an unordered list of operations, MongoDB:

- Groups and reorders the operations for enhanced performance.
- Further splits the groups when the number of operations cross 1000 in each group.

```
var bulk = db.items.initializeUnorderedBulkOp();
        bulk.insert( { _id: 1, item: “pen", available: true, soldQty:700} );
        bulk.insert( { _id: 2, item: “pencil", available: false, soldQty:900} );
        bulk.insert( { _id: 3, item: “books", available: true, soldQty: 600 } );
        bulk.execute( );
```

Performing Un-ordered Bulk Insert

simplilearn

This demo will show the steps to perform an unordered bulk insert in MongoDB.
Please refer to e-learning videos for this demo.

Inserts: Internals and Implications

Process of executing an Insert operation is as follows:

- Language drivers convert the data structure into Binary JSON (BSON) before sending to the database.
- The database looks for '_id' key and confirms that the document has not exceeded 16MB.
- The document is saved to the database without any changes.

MongoDB does the following before sending data to the database:

- Check for invalid data
- Checks UTF-8 compliance of strings
- Filter unrecognized data types

MongoDB is not vulnerable to traditional injection attacks because it does not perform any code execution on inserts.

Performing an Insert Operation

This demo will show the steps to perform an insert operation in MongoDB. Please refer to e-learning videos for this demo.

Retrieving the documents

MongoDB queries define the criteria or conditions for document retrieval. A query may also include a projection that defines the fields that match the document fields to return. You can modify queries to impose limits, skips, and sort orders.

```
db.items.find( {available: true },{item:1,} ).limit(5)
```

Specify Equality Condition

The FindOne() method finds the first record in a document. The pretty () method helps get properly formatted results. The query given below selects all documents in a collection displays the result in proper format.

```
db.items.find().pretty()
```

An empty ({}) query document selects all documents in the collection. The query given below finds all documents in a collection.

```
db.items.find( {} )
```

=

```
db.items.find()
```

To specify an equality condition, use the query document { <field>: <value>}. The query below retrieves all documents having the available field value as true.

```
db.items.find( {available: true } )
```

Retrieving Documents by Find Query

This demo will show the steps to retrieve documents from MongoDB collection by using the find() query. Please refer to e-learning videos for this demo.

$in, $or , and “AND” Conditions

You can specify query conditions using the following query operators:

- **$in**: Queries a variety of values for a single key.

```
db.items.find( {available : { $in: [true, false ] } } )
```

- **$or**: Queries similar values as $in operator. However, it is recommended to use the $in operator when performing equality checks on the same field.
- **AND**: A compound query can specify conditions for more than one field in the collection’s documents.

```
db.items.find( {available: true, soldQty: { $lt: 900 } }
)
```

$or Operator

Following are the functions of the $or operator:

- Returns the value true when any of its expressions evaluate to true or accepts any argument expressions.
- Defines a compound query where each clause are joined with a logical OR conjunction.

```
db.items.find({ $or: [ {soldQty : { $gt: 500 } }, { available:true } ] })
```

Specify AND/OR Conditions

The query example given below selects all documents in the collection where:

- The value of the available field is true
- The soldQty has a value greater than 200
- The value of the item field is “Book”

```
db.items.find({ available:true,$or: [ {soldQty : { $gt: 200 } }, {item: “Book” } ]})
```

The “$not” is a metaconditional operator. You can apply a $not to any other criteria. In the example given below, the “$mod” queries the keys whose values, when divided by the first given value, shows a remainder of the second value.

```
db.items.find({“_id" : {"$not" : {"$mod" : [4, 1]}}})
```

Retrieving Documents by Using FindOne, AND/OR Conditions

simplilearn

This demo will show the steps to retrieve documents in MongoDB by using the conditions : FindOne, And, OR. Please refer to e-learning videos for this demo.

Regular Expression

Regular expressions string matches flexible items and performs case insensitive matching.

```
db.items.find({item:/pe/i})
```

You can modify the regular expression to search for any variation.

```
db.items.find({item: /Pen?/i})
```

Array Exact Match

Equality matches can specify a single element in the array to match. If the array contains minimum one element with the specified value, these specifications match. The example below queries for documents that contains an array country_codes that contains 5 as one of its elements.

```
db.items.find( {country_codes: 5 } )
```

In the example below, the query uses the dot notation.

```
db.items.find( { 'country_codes.0': 1 } )
```

Array Projection Operators

MongoDB provides three projection operators—$elemMatch, $slice, and $. The operation given below uses the $slice projection operator.

```
db.items.find( { _id: 5 }, {country_codes : { $slice: 2 } } )
```

$elemMatch, $Slice, and $ are used to return a subset of elements for an array key. The operation given below uses $elemMatch to get that document in which the country_codes array value is ($gte) and ($lte) to 6.

```
db.items.find( {country_codes : { $elemMatch: { $gte: 3, $lte: 6 } } }
)
```

Retrieving Documents for Array Fields

This demo will show the steps to retrieve documents for array fields.
Please refer to e-learning videos for this demo.

$Where Query

The '$Where' clause allows you to do the following:

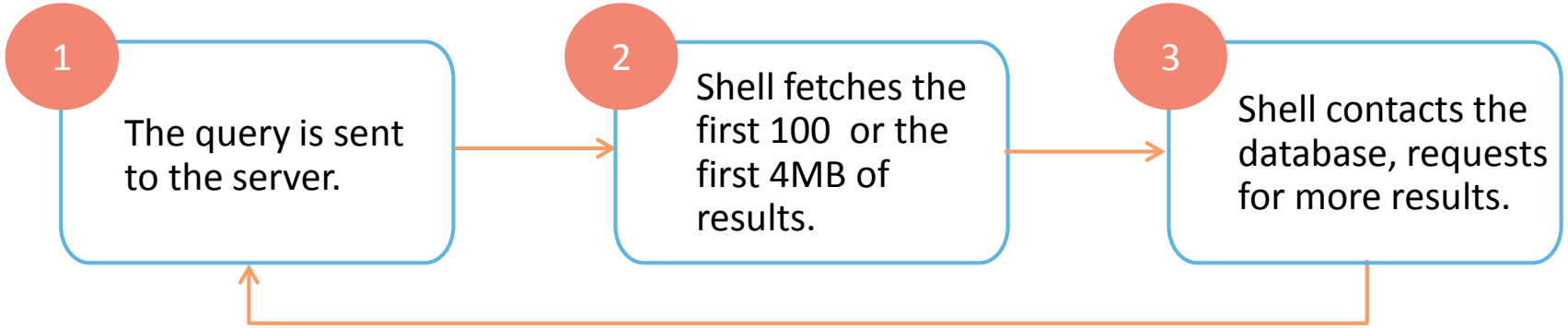
- Represent the queries that key/value pairs fail to represent.
- Perform any logical execution within a query.
- Compare the values of two keys in a document.

```
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

```
db.foo.find({"$where" : function () {
                ... for (var current in this) {
                ... for (var other in this) {
                … if (current != other && this[current] == this[other]) {
                ... return true;
                }} }
                .. return false;
                ... }});
db.foo.find({"$where" : "this.x + this.y == 10"})
> db.foo.find({"$where" : "function() { return this.x + this.y == 10; }"})
```

Cursor

Cursors allow you to control a query output by limiting the number of results, skipping some results, and sorting results. To create a cursor with the shell, add documents into a collection, perform a query on the documents, and then allocate the results to a local variable, such as "var".



Cursor (contd.)

Cursors allow you to control a query output by limiting the number of results, skipping some results, and sorting results. To create a cursor with the shell, add documents into a collection, perform a query on the documents, and then allocate the results to a local variable, such as "var".

```
var cursor = db.collection.find();
 while (cursor.hasNext()) {
... obj = cursor.next();
... // do stuff
... }
> var cursor = db.people.find();
> cursor.forEach(function(x) {
... print(x.name);
... });
db.c.find().skip(3)
db.c.find().sort({username : 1, age : -1})
```

Cursor (contd.)

Before sending a query to the database, you must include the following three options.

Limit

Set a limit to the number of results fetched. Example: To limit the number of documents found to three, use the query—db.items.find().limit(3)

Skip

Skip few documents from the result. Example: To skip the first three documents, use the query—db.c.find().skip(3)

Sort

Sort the results in ascending or descending manner. To sort results by name in the ascending and age in the descending manner, use the query —db.c.find().sort({username : 1, age : -1})

Retrieving Documents Using Cursor

This demo will show the steps to retrieve documents from MongoDB using cursor. Please refer to e-learning videos for this demo.

Pagination

To display 25 results sorted by price from high to low per page, perform the query given below.

```
db.stock.find({"desc" : “mobile"}).limit(25).sort({"price" : -1})
```

To display the Next Page for more results, use the second query given below.

```
db.stock.find({"desc" : “mobile"}).limit(25).skip(25).sort({"price" : -1})
```

Pagination : Avoiding Larger Skips

Pagination lets you control the number of documents returned by the find() query. To display the first page of a query result in the descending chronological order, use the query given below.

```
var page1 = db.foo.find().sort({"date" : -1}).limit(100)
```

For pagination without using a skip, use the query given below.

```
var latest = null;
// display first page
while (page1.hasNext()) {
latest = page1.next();
display(latest);
}
// get next page
var page2 = db.foo.find({"date" : {"$gt" : latest.date}});
page2.sort({"date" : -1}).limit(100);
```

Advance query option

The following options help in adding arbitrary options to queries.

$maxscan: Integer	This specifies the maximum number of documents that are scanned for a query.
$min: Document	This starts the criteria for querying.
$max: Document	This ends the criteria for querying.
$hint: Document	This tells the server which index to use for the query.
$explain: Boolean	Does not perform the actual query but explains how the query will be executed.
$snapshot: Boolean	This forces the query to use the index of the _ID field.

Update Operation

The db.collection.update() method in Mongo DB modifies existing documents in a collection. It identifies the documents that needs update and options that affect its behavior.

Operations performed by an update are atomic within a single document. This method has the following parameters:

- An update condition
- An update operation
- An options document

You need to use a structure and syntax for specifying the update condition and use ‘multi:true’ to update multiple documents.

Updating Documents in MongoDB

This demo will show the steps to update documents in MongoDB.
Please refer to e-learning videos for this demo.

$SET

MongoDB provides update operators, such as $set to modify values to change a field value. You need to perform the following steps to use a $set modifier.

1. **Use update operators to change field values**

```
db.items.update({ item:"Book"},{$set: {category: 'NoSQL',details: {ISDN: "1234",publisher:"XYZ"}}});
```

2. **Update an embedded field**

```
db.items.update({ item: “Pen" },{ $set: { "details.model": "14Q2" } })
```

3. **Update multiple documents**

```
db.items.update({ item: “Pen" },{$set: { category: "stationary" },$currentDate: { lastModified: true }},{ multi: true })
```

Updating Embedded Documents in MongoDB

simplilearn

This demo will show the steps to update an embedded document in MongoDB.
Please refer to e-learning videos for this demo.

Updating Multiple Documents in MongoDB

simplilearn

This demo will show the steps to update multiple documents in MongoDB.
Please refer to e-learning videos for this demo.

$Unset and $inc Modifiers

A $inc operator is used for incrementing and decrementing numbers. The query given below increments the value of the soldQty field by 1 for “Pencil”.

```
db.items.update({“item" : “Pencil"}, {"$inc" : {“soldQty" : 1}})
```

The value of the "$inc" key must be a number because you cannot increment a non-numeric value. To remove the soldQty field for all the documents where its value is greater than 700, use the query given below.

```
> db.items.update( {soldQty: { $gt: 700} },{ $unset: {soldQty: “1000" } },{ multi: true
})
```

$inc modifier to increment and decrement

This demo will show the steps to use the $inc modifier to increment and decrement a field value. Please refer to e-learning videos for this demo.

Replacing Existing Document with New Document

This demo will show the steps to replace existing documents with new documents in MongoDB. Please refer to e-learning videos for this demo.

$Push and $addToSet

The $Push function allows you to add new elements into an array field. The $push function does one of the following:

- Adds the element at the end of an array if the array already exists
- Creates an array field and insert the element into that field

The query given below allows you to use the $push function to an ingredients key.

```
db.user.update({“item" : “Pencil”}, {$push : {“ingredients" :  {“wood":"California cedar",“graphite":“mixture of natural graphite and chemicals"}}})
```

When adding another email address, use the “$addToSet” modifier to prevent duplicate.

Positional Array Modifications

You can modify the array values in two ways:

- By position
- By using the position operator (“$”)

To increment the votes for the first comment in a blog post, use the command given below:

```
db.blog.update({"post" : post_id},{"$inc" : {"comments.0.votes" : 1}})
```

The positional operator ("$“)helps identify the arrays matching the query document and updates them. The command given below shows how a positional operator updates an author's name in an existing document for the blog collection.

```
db.blog.update({"comments.author" : "John"}, {"$set" : {"comments.$.author" :
"Jim"}}))
```

Adding Elements into Array Fields

This demo will show the steps to add elements into array fields in MongoDB. Please refer to e-learning videos for this demo.

Adding Elements to Array Fields Using AddToSet

simplilearn

This demo will show the steps to add elements into existing array fields in MongoDB by using AddToSet modifiers. Please refer to e-learning videos for this demo.

Performing AddToSet

simplilearn

This demo will show the steps to use AddToSet to ignore duplicate elements to be inserted into array fields. Please refer to e-learning videos for this demo.

Upsert

An upsert is a special kind of update that does the following:

- Updates a document if it matches an update criteria are found.
- Inserts new documents into the collection if no matching criteria is found.

The command given below is an example of upsert operation.

```
db.items.update({“item" : “Bag"}, {"$inc" : {“soldQty" : 1}}, {upsert:true})
```

To update all the documents that matches the query criteria, you can use true as the fourth parameter.

Below is an example of a multi update command.

```
db.users.update({item: "10/22/1978"},{$set : {gift : "Happy Birthday!"}}, false, true)
```

Removing Documents

You can remove documents from MongoDB in the following way:

- To remove data from a collection use the command given below.

```
db.courses.remove()
```

- To remove documents from the items collection where the value for "item" is ”Bag”, use the command given below.

```
db.items.remove({“item" : “Bag”)
```

- To remove a single document, call the remove() method with the two parameters.

```
db.items.remove({" item" : “Bag},1)
```

- To delete all documents from a collection, use the drop() method.

```
db.courses.drop()
```

Performing Upsert and Remove Operation

This demo will show the steps to perform an upsert and remove operation in Mongo DB. Please refer to e-learning videos for this demo.

Quiz

QUIZ 1	Which of the following method either inserts a document or update the existing document?

a. Insert()

b. Save()

c. Update()

d. Bulk Insert

QUIZ 1	Which of the following method either inserts a document or update the existing document?

a. Insert()

b. Save()

c. Update()

d. Bulk Insert

The correct answer is **b**.

Explanation: The save() method either updates an existing document or inserts a new document.

QUIZ 2	Which of the following operator is used when you have more than one possible value to match for a single key?

a. Cursor

b. $or

c. $in

d. $all

QUIZ 2

Which of the following operator is used when you have more than one possible value to match for a single key?

a. Cursor

b. $or

c. $in

d. $all

The correct answer is **c**.

Explanation: $in is used to query for a variety of values for a single key.

QUIZ 3	Which of the following operator allows you to execute arbitrary JavaScript as part of your query?

a. $where

b. $Slice

c. $or

d. $elemMatch

QUIZ 3	Which of the following operator allows you to execute arbitrary JavaScript as part of your query?

a. $where

b. $slice

c. $or

d. $elemMatch

The correct answer is **a**.

Explanation: The "$where" clauses allow you to execute arbitrary JavaScript as part of your query.

QUIZ 4

What is the use $hint option in a wrapped query?

a. Ensure that the query results are a consistent snapshot from the point in time.

b. Specify the maximum number of documents that should be scanned for the query.

c. Get an explanation of how the query will be executed.

d. Tell the server which index to use for the query.

QUIZ 4

What is the use $hint option in a wrapped query?

a. Ensure that the query results are a consistent snapshot from the point in time.

b. Specify the maximum number of documents that should be scanned for the query.

c. Get an explanation of how the query will be executed.

d. Tell the server which index to use for the query.

The correct answer is **d**.

Explanation: The use of $hint in a wrapped query is to tell the server which index to use.

Summary

Here is a quick recap of what was covered in this lesson:

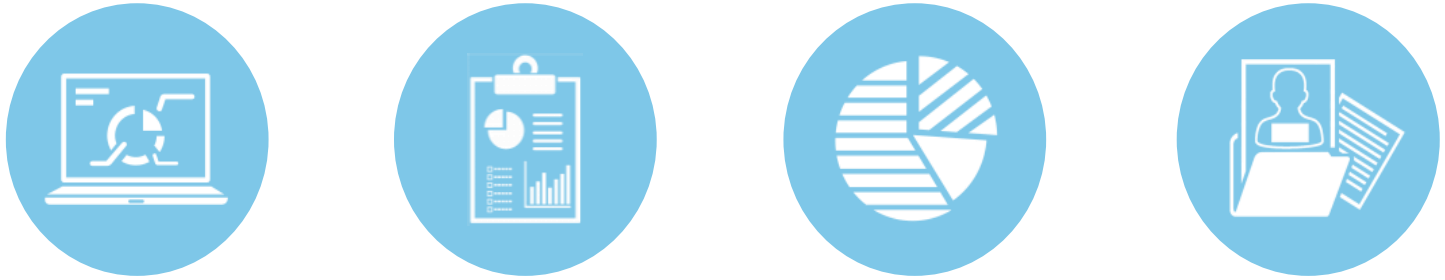
- Data modifications in Mongo DB involve creating, updating, or deleting data.
- Using batch insert, you can send hundreds or thousands of documents in a batch at a time.
- MongoDB divides an ordered operations into groups containing a maximum of 1000 operations.
- An unordered bulk inserts allow write operations in parallel in a random order.
- MongoDB allows you to specify conditions using query operators, such as $in, And, $or.
- Cursors in MongoDB allows you to modify queries to impose limits, skips, and sort orders.
- You can use pagination to customize the display results for a search query.

This concludes ‘CRUD Operations in MongoDB.’

The next lesson is ‘Indexing and Aggregation in MongoDB.’

MongoDB Developer

Indexing and Aggregation



After completing this lesson, you will be able to:

- Explain how to create unique, compound, sparse, text, and geospatial indexes in MongoDB
- Explain the process of checking the indexes used by MongoDB when retrieving the documents from the database
- Identify the steps to create, remove, and modify indexes
- Explain how to manage indexes by listing, modifying, and dropping
- Identify different kinds of aggregation tools available in MongoDB
- Explain how to use MapReduce to perform complex aggregation operations in MongoDB

Introduction to Indexing

Indexes are data structures that store collection's data set in a form that is easy to traverse. Indexes help perform the following functions:

- Execute queries and find documents that match the query criteria without a collection scan.
- Limit the number of documents a query examines.
- Store field value in the order of the value.
- Support equality matches are range-based queries.

MongoDB indexes are similar to the indexes in any other databases.

Types of Index

MongoDB supports the following index types for querying:

- **Default _id**: Each MongoDB collection contains an index on the default _id field.
- **Single Field**: For single-field index and sort operations, MongoDB can traverse the indexes either in the ascending or descending order.
- **Compound Index**: MongoDB supports user-defined indexes, such as compound indexes for multiple fields.
- **Multikey Index**: Used for indexing array data.
- **Geospatial Index**: Uses 2d indexes and 2d sphere indexes.
- **Text Indexes**: Searches data string in a collection.
- **Hashed Indexes**: MongoDB supports hash based sharding and provides hashed indexes.

Single Field Index

MongoDB supports indexes on any document field in a collection. By default, the _id field in all collections have indexes. Moreover, applications and users add indexes for triggering queries and performing operations.

MongoDB supports both, single field or multiple field indexes based on the operations the index-type performs.

```
db.items.createIndex( { “item" : 1 } )
```

Single Field Index

MongoDB supports indexes on any document field in a collection. By default, the _id field in all collections have indexes. Moreover, applications and users add indexes for triggering queries and performing operations.

MongoDB supports both, single field or multiple field indexes based on the operations the index-type performs.

```
db.items.createIndex( { “item" : 1 } )
```

Single Field Index on Embedded Document

You can index top level fields within a document. Similarly, you can create indexes within embedded document fields.

```
{ "_id" : 3, "item" : "Book", "available" : true, "soldQty" : 144821, "category" : "NoSQL", "details" : { "ISDN" : "1234",
                        "publisher" : "XYZ Company" }, "onlineSale" : true
            }
```

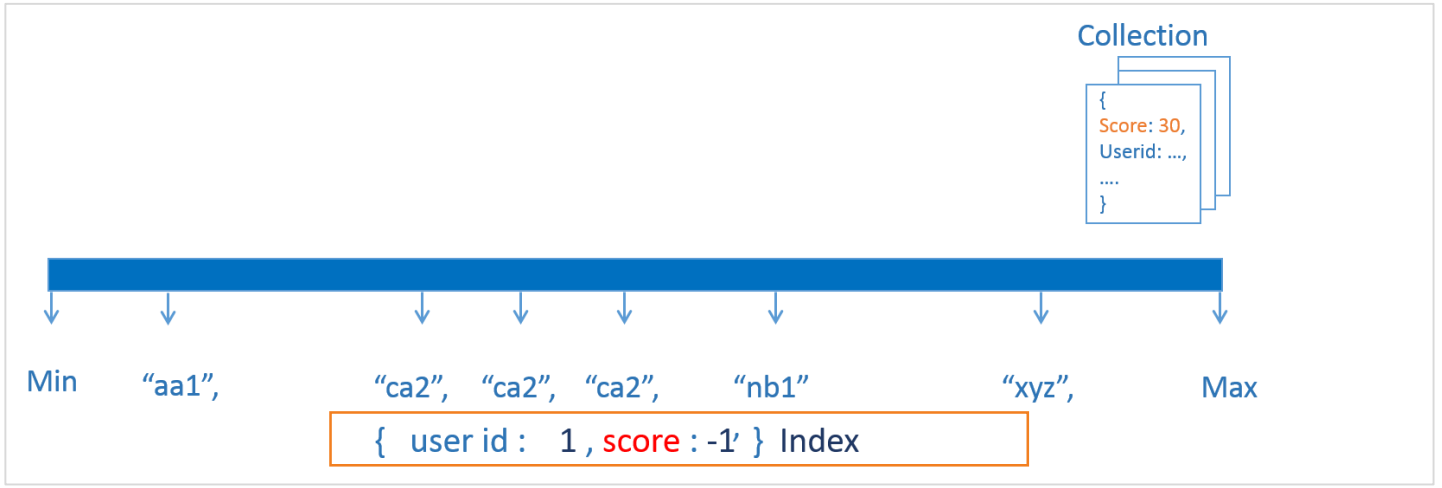
Use the query given below to create an index on the ISDN field and an embedded document.

```
db.items.createIndex( {details.ISDN: 1 } )
```

Compound Indexes

A compound index in Mongo DB contains multiple single field indexes separated by a comma.

```
db.products.createIndex( { "item": 1, "stock": 1 } )
```



MongoDB limits the fields of a compound index to a maximum of 31.

Index Prefixes

Index prefixes are created by taking different combination of fields and start from the first field. For example, consider the compound index given below.

```
{ "item": 1, “available”:1, "soldQty“:1}
```

MongoDB cannot efficiently support the query on the item and “soldQty” fields by using index prefixes.

The item field is a part of the compound index and the index prefixes and therefore should be used in the find query of the index.

Sort Order

The sort operations help retrieve documents based on the sort order in an index. Following are the characteristics of a sort order:

- If sorted documents cannot be obtained from an index, the results will get sorted in the memory.
- Sort operations executed using an index show better performance than those executed without using an index.
- Sort operations performed without an index gets terminated after exhausting 32 MB of memory.
- Indexes store field references in the ascending or descending sort order.
- Sort order is not important for single-field indexes because MongoDB can traverse the index in either direction.
- Sort order is important for compound indexes because it helps determine if the index can support a sort operation.

Multi-Key Indexes

When indexing a field containing an array value, MongoDB creates separate index entries for each array component. MongoDB lets you construct multi-key indexes for arrays holding scalar values, such as strings, numbers, and nested documents.

To create a multi-key index, use the method given below.

```
db.coll.createIndex( { <field>: < 1 or -1 > } )
```

If the indexed field contains an array, MongoDB automatically decides to:

- Create a multi-key index
- Not create a multi-key index

Multi-Key Indexes

When indexing a field containing an array value, MongoDB creates separate index entries for each array component. MongoDB lets you construct multi-key indexes for arrays holding scalar values, such as strings, numbers, and nested documents.

To create a multi-key index, use the method given below.

```
db.coll.createIndex( { <field>: < 1 or -1 > } )
```

If the indexed field contains an array, MongoDB automatically decides to:

- Create a multi-key index
- Not create a multi-key index

Compound Multi-Key Indexes

In compound multi-key indexes, each indexed document can have maximum one indexed field with an array value. When more than one field contain an array value, compound multi-key indexes cannot be created.

Given below is an example of document structure.

```
{ _id: 1, product_id: [ 1, 2 ], retail_id: [ 100, 200 ], category: "both fields are arrays" }
```

A shard key index and a hashed index cannot be a multikey index.

Hashed Indexes

The hashing function does the following:

- Combines all embedded documents.
- Computes hashes for all field values.
- Supports sharding, uses a hashed shard key to shard a collection, and ensures an even data distribution.
- Supports equality queries, however, range queries are not supported.

You cannot create unique or compound index by taking a field whose type is hashed. However, you can create a hashed and non-hashed index for the same field.

To create a hashed index, use the operation given below.

```
db.items.createIndex( { item: "hashed" } )
```

TTL Indexes

Total Time to Live (TTL) indexes can be created by combining db.collection.createIndex() and “expireAfterSeconds”.

To create a TTL index, use the operation given below.

```
db.eventlog.createIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
```

TTL indexes have the following limitations:

- Not supported by compound indexes and the _id field does not support TTL indexes
- Cannot be created on a capped collection
- Does not allow createIndex() to change the value of “expireAfterSeconds” of an existing index

To change a non-TTL single-field index to a TTL index, drop the index and recreate the index with the “expireAfterSeconds” option.

Unique Indexes

Unique indexes can be created by using the db.collection.createIndex() method and set the unique option to true. To create a unique index on the item field of the items collection, execute the operation given below.

```
db.items.createIndex( { “item": 1 }, { unique: true } )
```

If a unique index has no value, the index stores a null value for the document. Because of this unique constraint, MongoDB permits only one document without the indexed field. For more than one document with a valueless or missing indexed field, the index build process fails.

To filter null values in a document and avoid error, combine the unique constraint with the arse index.

Sparse Indexes

Sparse indexes manage documents with indexed fields and ignores documents which do not contain any index field.

To create a sparse index, use the db.collection.createIndex() method and set the sparse option to true.

```
db.addresses.createIndex( { "xmpp_id": 1 }, { sparse: true } )
```

When a sparse index returns an incomplete index, then MongDB does not use that index unless it is specified in the hint method.

```
{ x: { $exists: false } }
```

An index combining sparse and unique indexes does not allow duplicate field values for a single field.

Demo—Create Compound, Sparse, and Unique Indexes

This demo will show the steps to create compound, sparse, and unique indexes.

Please refer to e-learning videos for this demo.

Text Indexes

Text indexes in MongoDB helps search for text strings in documents of a collection. To access text indexes, trigger a query using the $text query operator.

When you create text indexes for multiple fields, specify the individual fields or use the wildcard specifier ($**).

To create text indexes on the subject and content fields, perform the query given below.

```
db.collection.createIndex({subject: "text",content: "text"})
```

The wildcard specifier ($**) indexes all fields containing string content. The example given below indexes any string value available in each field of each document.

```
db.collection.createIndex({ "$**": "text" },{ name: "TextIndex" })
```

Demo—Create Single Field and Text Index

simplilearn

This demo will show the steps to create single field and text indexes in MongoDB.
Please refer to e-learning videos for this demo.

Text Search

MongoDB supports various languages for text search. The text indexes use simple language-specific suffix stemming instead of language-specific stop words, such as “the”, “an”, “a”, “and”.

The query given below performs a text search for the item field “customer_info” with Spanish as the default language.

```
db.customer_info.createIndex({“item”: “Text”},{ default_language: "spanish"})
```

The text index and the $text operator supports the following:

- Two-letter language codes defined in ISO 639-1
- Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish, and Turkish

A compound text index cannot include special index types, such as multi-key or geospatial Index fields.

Index Creation

During index creation, operations on a database are blocked and the database becomes unavailable for any read or write operation. The read or write operations on the database queue allow the index building process to complete.

To make MongoDB available even during an index build process, use the command given below.

```
db.items.createIndex( {item:1},{background: true})

db.items.createIndex({category:1}, {sparse: true, background: true})
```

By default, background is false for building MongoDB indexes.

Index Creation (contd.)

If you perform any administrative operations when MongoDB is creating indexes in the background for a collection, you will receive an error.

The index build process at the background:

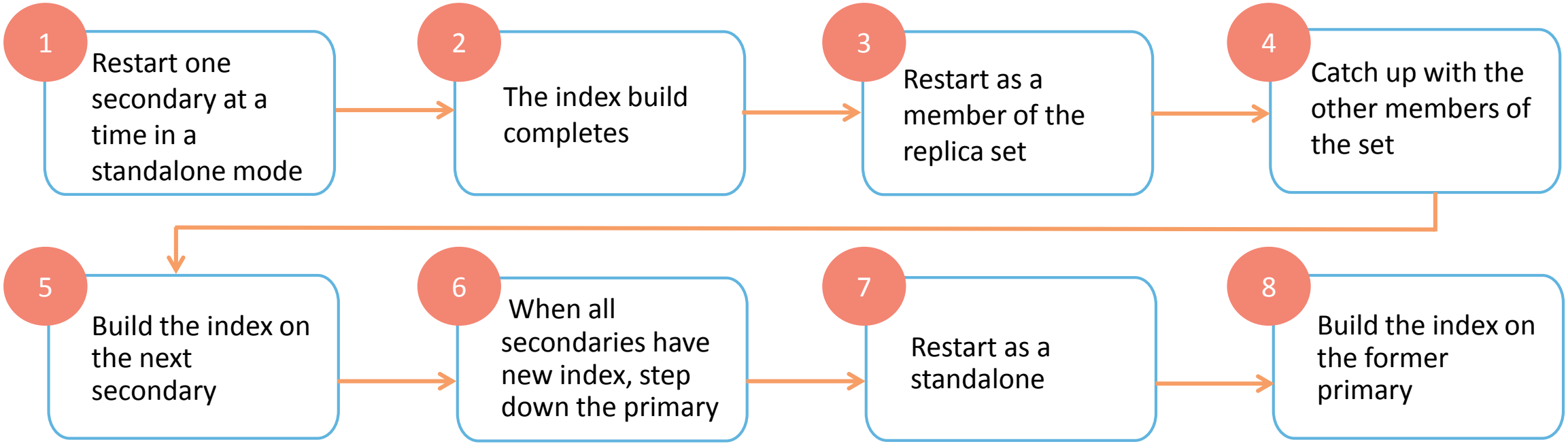
- Uses an incremental approach and is slower than the normal “foreground” process.
- Depends on the size of the index for its speed.
- Impacts database performance.

To avoid any performance issues, use:

- getIndexes() to ensure that your application checks for the indexes at the start up.
- Equivalent method for your driver and ensure it terminates an operation if the proper indexes do not exist.
- Separate application codes and designated maintenance windows.

Index Creation on Replica Set

Background index operations on a secondary replica set begin after the index build completes in the primary. To build large indexes on secondaries perform the following steps:



Use the command below to specify a name for an index.

```
db.products.createIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

Remove Indexes

Use the following methods to remove indexes.

dropIndex() Method

To remove an ascending index on the item field in the items collection:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

db.collection.dropIndexe() Method

To remove all indexes barring the _id index from a collection, use the command:

```
db.collection.dropIndexes()
```

Modify Indexes

To modify an index, perform the following steps.

1

Drop Index: Execute the query given below to return a document showing the operation status.

```
db.orders.dropIndex({ "cust_id" : 1, "ord_date" : -1, "items" : 1 })
```

2

Recreate the Index: Execute the query given below to return a document showing the status of the results.

```
db.orders.createIndex({ "cust_id" : 1, "ord_date" : -1, "items" : -1 })
```

Demo—Drop and Index from a Collection

This demo will show the steps to drop indexes for a collection. Please refer to e-learning videos for this demo.

Demo—Drop and Index from a Collection

This demo will show the steps to drop indexes for a collection. Please refer to e-learning videos for this demo.

Rebuild Indexes

To rebuild all indexes of a collection, use the db.collection.reIndex method. This will drop all indexes including _id and rebuild all indexes in a single operation. You can use the following commands when rebuilding indexes:

- **db.currentOp()**—Type this command in the mongo shell to view the indexing process status.
- **db.killOp()**—Type this command in the mongo shell to abort an ongoing index build process.

You cannot abort a replicated index built on the secondary replica set.

Listing Indexes

All indexes of a collection and a database can be listed. To get a list of all indexes of a collection, use the db.collection.getIndexes() or a similar method.

To list all indexes of collections, use the operation given below in the mongo shell.

```
db.getCollectionNames().forEach(function(collection) {
indexes = db[collection].getIndexes();
print("Indexes for " + collection + ":");
printjson(indexes);
});
```

Demo—Retrieve Indexes for a Collection and Database

This demo will show the steps to retrieve indexes for a collection and a database. Please refer to e-learning videos for this demo.

Measure Index Use

Query performances indicate index usage. MongoDB provides the following methods to observe index use for your database:

- **The explain() method**—used to print information about query execution. Returns a document that explains the process and indexes used to return a query.
- **db.collection.explain() or the cursor.explain()** —helps measure index usages.

Demo—Use Mongo Shell Methods to Monitor Indexes

This demo will show the steps to use different mongo shell methods to monitor the usage of indexes. Please refer to e-learning videos for this demo.

Control Index Use

To force MongoDB to use particular indexes for querying documents, you need to specify the index with the hint() method, which can be appended in the find() method.

The command given below queries a document whose item field value is “Book” and available field is “true”.

```
db.items.find({item: “Book”, available : true }).hint({item:1})
```

To view the execution statistics for a specific index, use the query below.

```
db.items.find({item: “Book”, available : true }).hint({item:1}).explain(“executionStats”)

db.items.explain("executionStats").find({item: “Book”, available : true }).hint( { item:1 } )
```

To prevent MongoDB from using any index, specify the $natural operator to the hint() method.

```
db.items.find({item: “Book”, available : true }).hint({$natural:1}).explain(“executionStats”)
```

Demo—Use the Explain, $Hint and $Natural Operators to Create Index

simplilearn

This demo will show you the steps to use the operators, explain, hint, and natural to create an index. Please refer to e-learning videos for this demo.

Index Use Reporting

MongoDB provides different metrics to report index use and operation.

These metrics are printed using the following commands.

- serverStatus:
  - scanned: Displays the documents that MongoDB scans in the index to carry out the operation
  - scanAndOrder: A boolean that is true when a query cannot use the order of documents in the index for returning sorted results
- collStats:
  - totalIndexSize: Returns index size in bytes
  - indexSizes: Explains the size of the data allocated for an index
- dbStats:
  - dbStats.indexes: Contains a count of the total number of indexes across all collections in the database
  - dbStats.indexSize: The total size in bytes of all indexes created on this database

Geospatial Index

MongoDB provides geospatial indexes for coordinating geospatial queries.

To find any location from your current location, you need to create a special index and search in two dimensions—longitude and latitude. A geospatial index is created using the createIndex() function. It passes "2d" or “2dsphere” as a value instead of 1 or -1.

To query geospatial data, you first need to create geospatial index. Use the command given below to create a geospatial index.

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

A compound index can include a 2dsphere index key in combination with non-geospatial index keys.

Demo—Create Geospatial Index

This demo will show the steps to create geospatial indexes in MongoDB.

Please refer to e-learning videos for this demo.

MongoDB's Geospatial Query Operators

The geospatial query operators in MongoDB lets you perform the following queries.

Inclusion Queries

- Return the locations included within a specified polygon.
- Use the operator $geoWithin. The 2d and 2dsphere indexes support this query.

Intersection Queries

- Return locations intersecting with a specified geometry.
- Use the $geoIntersects operator and return the data on a spherical surface.

Proximity Queries

- Return various points closer to a specified point.
- Use the $near operator that requires a 2d or 2dsphere index.

Demo—Use Geospatial Index in a Query

This demo will show the steps to use geospatial indexes in the find query of MongoDB. Please refer to e-learning videos for this demo.

$GeoWith Operator

The $geoWithin operator is used to query location data found within a GeoJSON polygon.

Use the syntax given below to use the $geoWith Operator.

```
db.<collection>.find( { <location field> :
{ $geoWithin :{ $geometry :{ type : "Polygon" ,coordinates : [ <coordinates> ]} } } } )
```

The example given below selects all points and shapes that exist entirely within a GeoJSON polygon.

```
db.places.find( { loc :{ $geoWithin :
{ $geometry :{ type : "Polygon" ,coordinates :[ [[ 0 , 0 ] ,[ 3 , 6 ] ,[ 6 , 1 ] ,[ 0 , 0 ]] ]} } } } )
```

Proximity Queries in MongoDB

Proximity queries return the points closest to the specified point and sort the results by its proximity to the specified point.

To perform a proximity query on the GeoJSON data points, you need to:

- Create a 2dsphere index
- Use the $near or $geonear operator

The $near operator uses the syntax given below:

```
db.<collection>.find( { <location field> :{ $near :{ $geometry :{ type : "Point" ,coordinates : [ <longitude> , <latitude> ] } ,
$maxDistance : <distance in meters>} } } )
```

The geoNear command uses the second syntax given below:

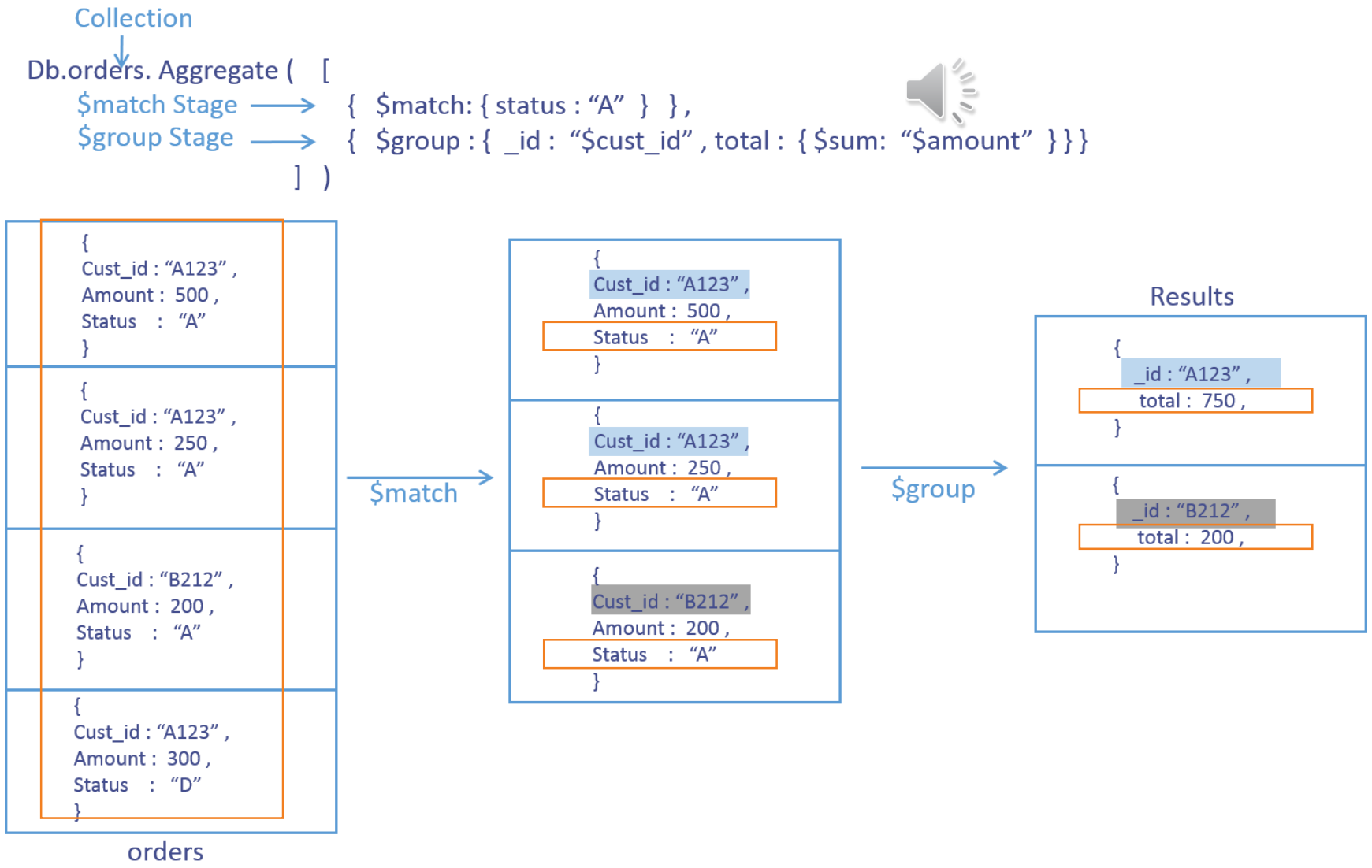
```
db.runCommand( { geoNear : <collection> ,near : { type : "Point" ,coordinates: [ <longitude>, <latitude> ] } ,
spherical : true } )
```

Aggregation

Aggregations process data sets and return calculated results. It is run on the mongod instance to simplify application codes and limit resource requirements. Following are the characteristics of aggregation:

- Uses collections of documents as an input and return results in the form of one or more documents.
- Is based on data processing pipelines. Documents pass through multi-stage pipelines and gets transformed into an aggregated result.
- The most basic pipeline stage in the aggregation framework provides filters that function like queries.
- The pipeline operations group and sort documents by defined field or fields.
- The pipeline uses native operations within MongoDB to allow efficient data aggregation and is the favoured method for data aggregation.

Aggregation (contd.)



Pipeline Operators and Indexes

The aggregate command in MongoDB, functions on a single collection and logically passes the collection through the aggregation pipeline. Use the $match, $limit, and $skip stages to optimize aggregate operations.

You may require only a subset of data from a collection to perform an aggregation operation. Therefore, use the $match, $limit, and $skip stages to filter the documents. The $match operation scans and selects only the matching documents in a collection when placed at the beginning of a pipeline.

Placing a $match pipeline stage followed by a $sort stage at the beginning of the pipeline is equivalent to a single query with a sort and can use an index. Place $match operators at the beginning of the pipeline if possible.

Aggregate Pipeline Stages

Documents are passed through the pipeline stages in a proper order one after the other. The various pipeline stages are:

$project

Adds new fields or removes existing fields and thus restructure each document in the stream. It returns one output document for each input document provided.

$match

Filters the document stream and allows only matching documents to pass into the next stage. For each input document, it returns one output document if there is a match or zero if no match is found.

$group

Groups documents based on the specified identifier expression and applies logic known as accumulator expression to compute the output document.

$sort

Rearranges the order of the document stream using specified sort keys. Provides one output document for each input document.

Aggregate Pipeline Stages (contd.)

Some more pipeline stages are:

$skip

Skips n number of documents and passes the remaining documents without any modifications to the pipeline. Returns either zero documents for the first n documents or one document.

$limit

Passes the first n number of documents without any modifications to the pipeline. Returns either one document for the first n documents or zero documents after the first n documents.

$unwind

Deconstructs an array field in the input documents to return a document for each element.

Aggregation Example

The aggregation operation given below returns all states with total population greater than 10 million.

```
db.zipcodes.aggregate( [{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 10*1000*1000 } } }] )
```

In this operation, the $group stage does the following:

- Groups the documents of the zipcode collection by the state field
- Calculates “thetotalPop” field for each state
- Returns an output document for each unique state

The aggregation operation given below returns user names sorted by the month of their joining.

```
db.users.aggregate([{ $project :{month_joined : { $month : "$joined" },name : "$_id",_id : 0}},{ $sort : { month_joined : 1 }
}
])
```

Demo—Use Aggregate Function

This demo will show the steps to use the aggregate pipeline framework in MongoDB. Please refer to e-learning videos for this demo.

MapReduce

MapReduce is a data processing model used for aggregation. To perform map-reduce operations, MongoDB provides the MapReduce database command.

A MapReduce operation consists of the following two phases:

- **Map stage**: Documents are processed and one or more objects are produced for each input document.
- **Reduce stage**: The outputs of the map operation are combined.

Optionally, there can be an additional stage to make final modifications to the result.

MapReduce can define a query condition to select the input documents, and sort and limit the results.

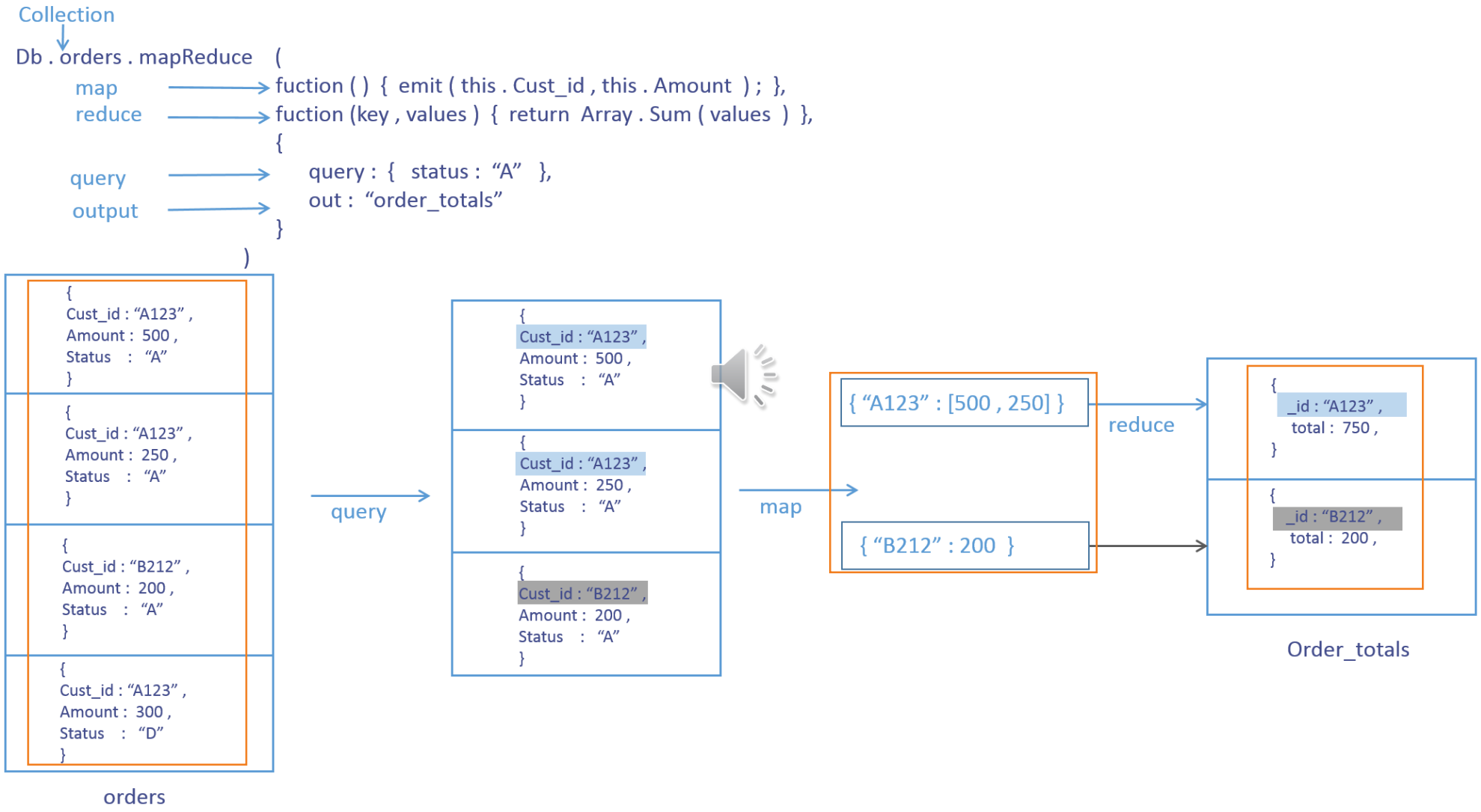
MapReduce (contd.)

The MapReduce function in MongoDB can be written as JavaScript codes. Typically, MapReduce operations:

- Accept documents as inputs, performs sort and limit functions, and then start the map stage. At the end of a MapReduce operation, the result is generated as documents which can be saved in the collection.
- Associate values to a key by using the custom JavaScript functions. If a key contains more than one mapped value, then this operation converts them into a single object, such as an array.

The use of custom JavaScript functions make the MapReduce operations flexible. The MapReduce operations use the custom JavaScript function to alter the results at the conclusion of the map and reduce operations may perform further calculations.

MapReduce (contd.)



Demo—Use MapReduce in MongoDB

This demo will show the steps to use the MapReduce function in MongoDB.

Please refer to e-learning videos for this demo.

Aggregation Operations

Aggregation operations manipulate data and return a computed result based on the input document and a specific procedure. It provides the following semantics for data processing:

Count

This command along with the two methods, count() and cursor.count() provide access to total counts in the mongo shell. The command given below counts all documents in the “customer_info” collection.

```
db.customer_info.count()
```

Distinct

This operation searches for documents matching a query and returns all unique values for a field in the matched document. The syntax given below is an example of a distinct operation.

```
db.customer_info.distinct( “customer_name" )
```

Demo—Use Distinct and Count Methods

This demo will show the steps to use the distinct and count methods in MongoDB.
Please refer to e-learning videos for this demo.

Aggregation Operations (contd.)

Group operations accept sets of documents as input which matches the given query, apply the operation, and then return array of documents with the computed results.

A group does not support sharded collection data. In addition, the results of the group operation must not exceed 16 megabytes.

The group operation shown below groups documents by the field ‘a’, where ‘a’ is less than three and sums the field count for each group.

```
db.records.group( {key: { a: 1 },cond: { a: { $lt: 3 } },reduce: function(cur, result) { result.count += cur.count },initial: { count: 0 }} )
```

Demo—Use the Group Function

This demo will show the steps to use the group function in MongoDB.

Please refer to e-learning videos for this demo.

Quiz

QUIZ 1	Which of the following method is used for listing all indexes of a collection?

a. getIndex()

b. listIndex()

c. getIndexes()

d. listIndexes()

QUIZ 1	Which of the following method is used for listing all indexes of a collection?

a. getIndex()

b. listIndex()

c. getIndexes()

d. listIndexes()

The correct answer is **c**.

Explanation: getIndexes() is used to list all the indexes created for a collection.

QUIZ 2	Which method do you use to verify the indexes MongoDB uses when executing a query?

a. explain()

b. hint()

c. $natural

d. $all

QUIZ 2	Which method do you use to verify the indexes MongoDB uses when executing a query?

a. explain()

b. hint()

c. $natural

d. $all

The correct answer is **a**.

Explanation: You can use the explain() method to see the query plan to check which indexes are being used.

QUIZ 3	Which among the following is the correct syntax to create an unique index on field name?

a. db.collection.createUniqueIndex({name:1})

b. db.collection.createUniqueIndex({name:1})

c. db.collection.createIndex({unique:true},{name:1})

d. db.collection.createIndex({name:1},{unique:true})

QUIZ 3	Which among the following is the correct syntax to create an unique index on field name?

a. db.collection.createUniqueIndex({name:1})

b. db.collection.createUniqueIndex({name:1})

c. db.collection.createIndex({unique:true},{name:1})

d. db.collection.createIndex({name:1},{unique:true})

The correct answer is **d**.

Explanation: unique:true should be passed as 2nd paramerter in the createIndex().

QUIZ 4

Which among the following statement is correct regarding MongoDB indexes?

a. MongoDB uses indexes for performing collection scan

b. MongoDB does not support sparse index

c. Indexes support the efficient execution of queries in MongoDB

d. Indexes in MongoDB are not similar to indexes in other database systems

QUIZ 4

Which among the following statement is correct regarding MongoDB indexes?

a. MongoDB uses indexes for performing collection scan

b. MongoDB does not support sparse index

c. Indexes support the efficient execution of queries in MongoDB

d. Indexes in MongoDB are not similar to indexes in other database systems

The correct answer is **c**.

Explanation: Indexes support the efficient execution of queries in MongoDB.

Summary

Here is a quick recap of what was covered in this lesson:

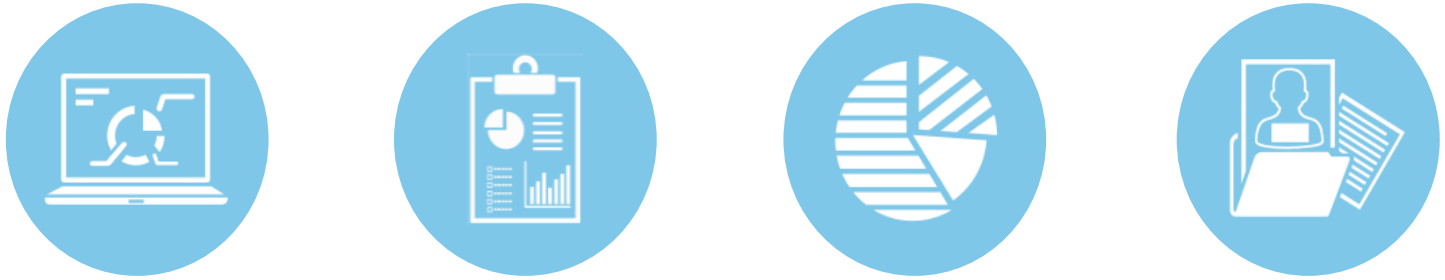
- Indexes are data structures that store data set in easily traversable form.
- Indexes help execute queries efficiently without performing a collection scan.
- MongoDB supports the following indexes—single field, compound, multikey, geospatial, text, and hashes.
- For fast query operation, the system RAM must be able to accommodate index sizes.
- You can create, modify, rebuild, and drop indexes.
- The geospatial indexes help query geographic location by specifying a specific point.
- The aggregation operations manipulate data and return a computed result based on the input and a specific procedure.
- Aggregation functions in MongoDB help query operations such as calculating total sum spent by a customer on online shopping site.

This concludes ‘Indexing and Aggregation in MongoDB’.

The next lesson is ‘Replication and Sharding in MongoDB’.

MongoDB Developer

Lesson 5: Replication and Sharding



After completing this lesson, you will be able to:

- Explain how the replication process works
- Explain the master-slave replication, replica sets, and replica set members
- Explain automatic failover
- Explain write concern, read preference, and tag sets
- Describe the replica set deployment architecture
- Explain how to enable sharding for database and collections
- List different types of sharding keys
- Explain when to use sharding and how to deploy a sharded cluster

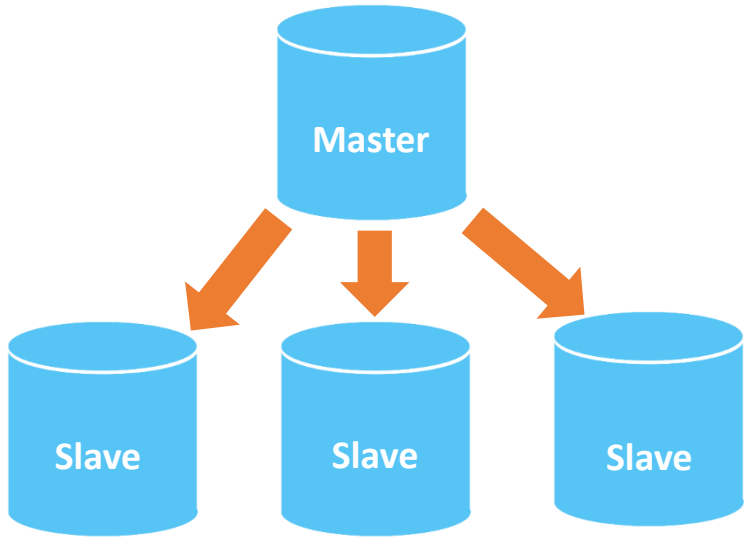
Introduction to Replication

The primary task of a MongoDB administrator is to set up a functioning replication in the production setting. Following are the benefits of replication:

- Increases data availability by creating redundancy.
- Stores multiple copies of data across different databases in multiple locations, and thus protects data when the database suffers any loss.
- Helps manage data in the event of hardware failure and any kind of service interruptions.
- Enhances read operations.
- Stores copies of these operations in different data centers to increase the locality and availability of data for distributed applications.

Master-Slave Replication

The Master-Slave Replication is the oldest mode of replication that MongoDB supports. In the earlier versions of MongoDB, the master-slave replication was used for failover, backup, and read scaling. However, in the newer versions, it is replaced by replica sets for most use cases.

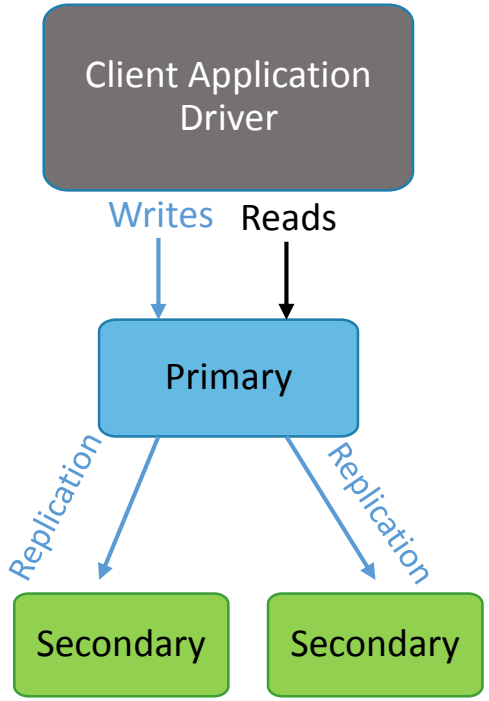


Replica sets are recommended for new production deployments to replicate data in a cluster rather than the master-slave replication.

Replica Set in MongoDB

A replica set consists of a group of mongod instances that host the same data set. The replica set functions as follows:

- The primary mongod receives all write operations and the secondary mongod replicates the operations from the primary.
- The primary node receives the write operations from clients.
- The primary logs any changes or updates to its data sets in its oplog.
- The secondaries replicate the oplog of the primary and apply all the operations to their data sets.
- When the primary becomes unavailable, the replica set nominates a secondary as the primary.



Replica Set in MongoDB (contd.)

An extra mongod instance can be added to a replica set to act as an arbiter. Following are some characteristics of an arbiter:

- Arbiters do not maintain a data set.
- Arbiter is the node which just participated in an election to select the primary node.
- Arbiters do not require a dedicated hardware.

Secondary members in a replica set asynchronously apply operations from the primary. These replica sets can function without some secondary members. As a result, all secondary members may not return the updated data to clients.

A primary can convert to a secondary or vice versa, however, an arbiter remains unchanged.

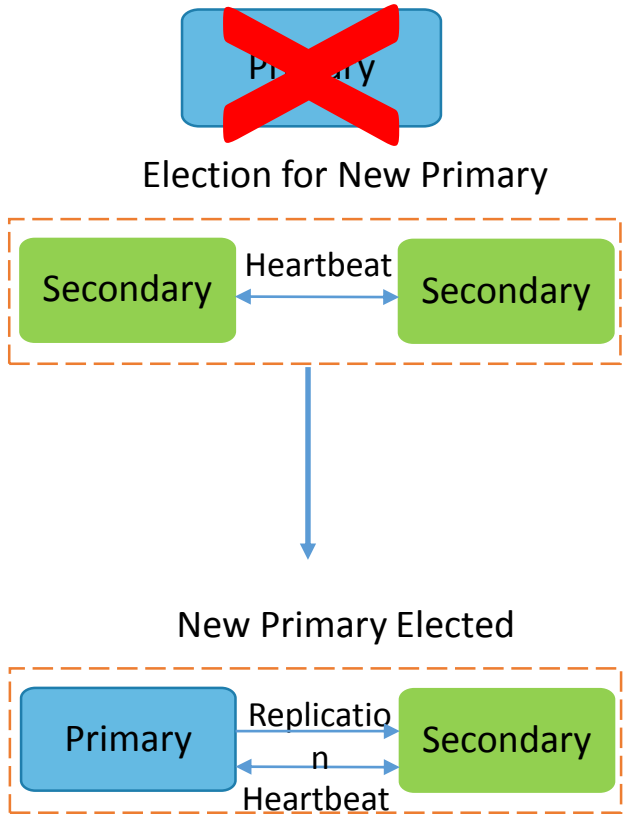
Automatic Failover

When the primary node of a replica set stops communicating with other members for more than 10 seconds:

- Another member is selected as the new primary.
- Selection happens through an election process.
- The secondary node that gets majority of votes becomes the primary.

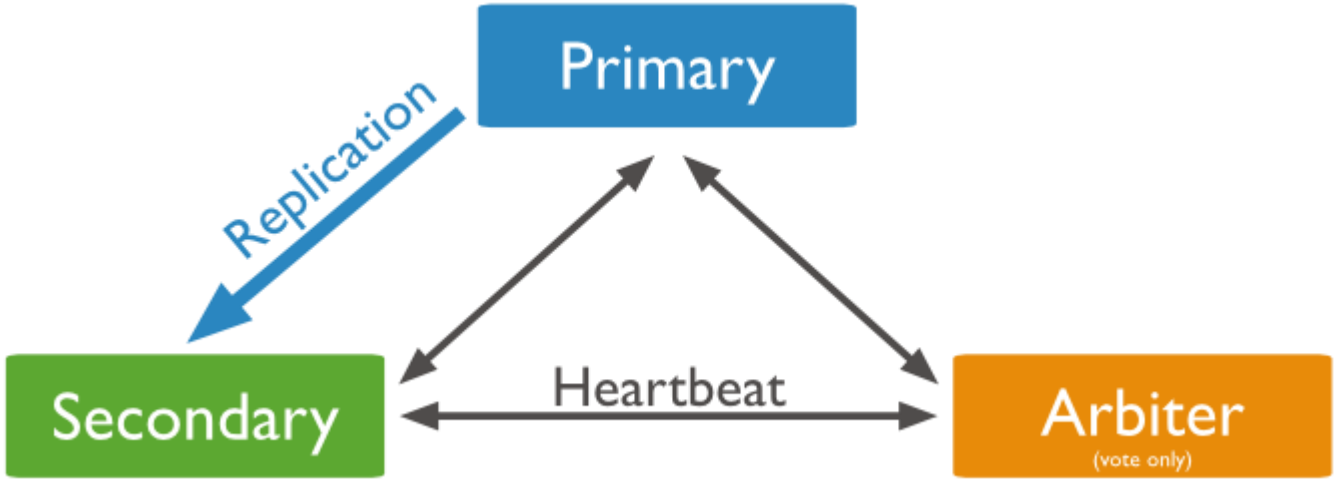
A replica set supports application needs in the following ways:

- Deploying a replica set in multiple data centers.
- Manipulating primary election by adjusting the priority of members.
- Supporting dedicated members for functions, such as reporting, disaster recovery, or backup.



Replica Set Members

A replica set can also have an arbiter. Arbiters do not replicate or store data but play a crucial role in selecting a secondary to take the place of the primary, when the primary becomes unavailable. A typical replica set contains a primary, secondary, and an arbiter.



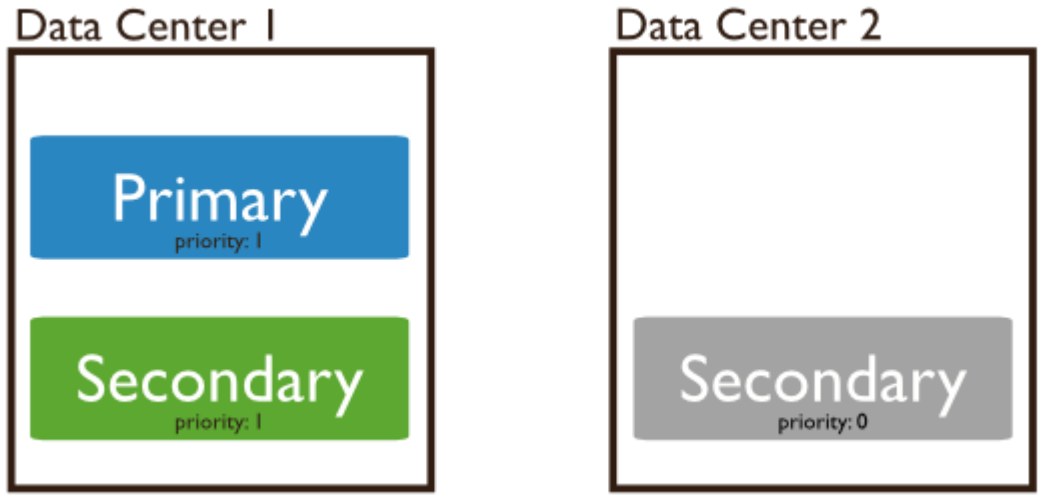
! A replica set in MongoDB version 3.0 can have maximum 50 members with only 7 members capable of voting.

Priority 0 Replica Set Members

A priority 0 member is a secondary member that cannot become the primary. The characteristics of a priority 0 are:

- Cannot trigger any election.
- Can maintain data set copies, accept and perform read operations, and elect a primary.

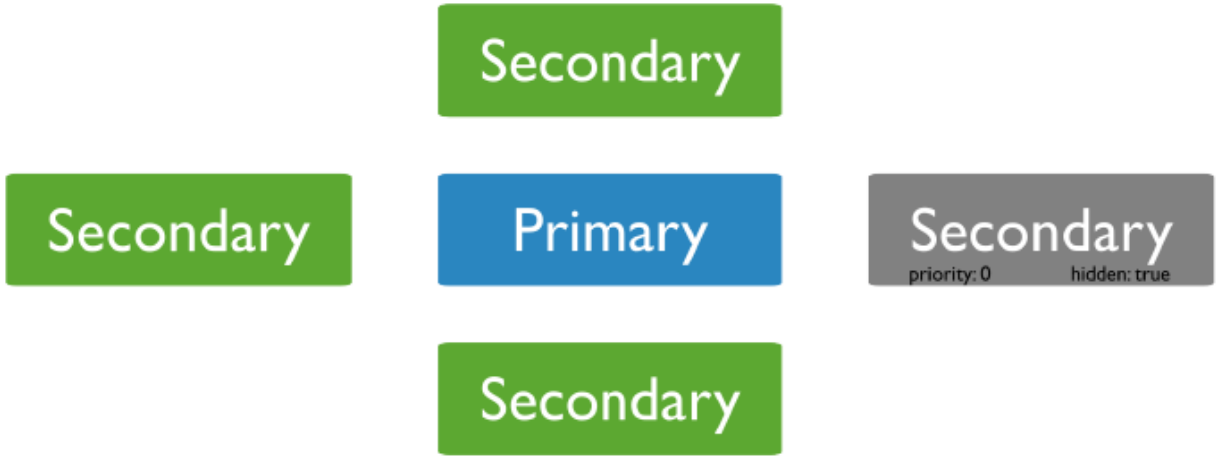
By configuring a priority zero member, you can prevent secondaries from becoming the primary. In a three-member replica set, one data center hosts both, the primary and a secondary, and a second data center hosts one priority zero member.



Hidden Replica Set Members

Hidden members of a replica set are invisible to the client applications. The characteristics of a hidden member are:

- They store a copy of the primary's data.
- They are priority 0 members, can elect a primary but cannot replace a primary.
- They are not given appropriate read and write rights.
- They can be used for dedicated functions like reporting and backup.



Delayed Replica Set Members

Delayed Replica Set Members are secondaries that copy data from the primary node's oplog file after certain interval or delay. Delayed replica set members reflect a previous version, or delayed state of the stored data set.

Delayed members perform a "roll backup" or run a "historical" snapshot of the data set. They help manage various human errors and recover from errors, such as unsuccessful application upgrade, and dropped databases and collections.

The characteristics of a delayed member are as follows:

- Must be a priority zero member
- Must be hidden and not be visible to applications
- Must participate in electing a primary

Delayed Replica Set Members (contd.)

You can configure a delayed secondary member with the settings given below:

- Priority value—zero
- Hidden value—true
- slaveDelay value—number of seconds to delay

To set an one hour delay, issue the operations given below:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

Demo—Start a Replica Set

This demo will show the steps to start a replica set in MongoDB. Please refer to e-learning videos for this demo.

Write Concern

The write concern ensures a successful write operation. The characteristics of a write concern are:

- The strength/weakness of a write concern determines the level of assurance.
- Write operations having weak write concerns may not succeed.
- When stronger write concerns are used in an operation, clients wait after sending the operation for MongoDB to confirm their success.

MongoDB provides different levels of write concerns to address varied application needs. For less critical operations, clients can manage the write concern to ensure faster performance rather than ensure persistence to the entire deployment.

Regardless write concern level or journaling configuration, MongoDB allows clients to read the inserted or modified documents before committing the modifications to the disk. As a result, if an application has multiple reader and writer, then MongoDB allows the reader to read the data of another writer even when data is still not committed to the journal.

Write Concern (contd.)

Some more characteristics of a write concern are:

- MongoDB modifies each document of a collection separately. For multi-document operations, MongoDB does not provide any multi-document transactions or isolation.
- When a standalone mongod returns a successful journaled write concern, the data is stored to the disk and becomes available after mongod restarts.
- Write operations in a replica set can become durable only after a write data gets replicated and committed to the journal on majority of the voting members of the replica set.
- MongoDB periodically commits the data to journal as defined by the “commitIntervalMs” parameter.

Write Concern Levels

MongoDB has the following levels of conceptual write concerns:

Unacknowledged

Does not acknowledge the received write operations or ignores it. Receives and handles network errors. The ability to detect and handle network errors depend on the networking configuration of the system.

Acknowledged

This is the default write concern for MongoDB. Acknowledges the receipt of a write operation. It confirms that the changes to the in-memory data view is applied. It can detect network, duplicate key, and other errors.

Journaling must be enabled to use a write concern.

Write Concern for a Replica Set

Replica sets have additional considerations for write concerns, which are as follows:

- The default write concerns require acknowledgement only from the primary member.
- A write concern acknowledged by a replica set ensures that the write operation spreads to the other members of the set.
- The default write concern confirms write operations only for the primary.
- The default write concerns can be overridden by specifying a write concern for each write operation.

```
db.products.insert({ item: "envelopes",qty:
    100, type: "Clasp" },
    { writeConcern: { w: 2, wtimeout: 5000
} })
```

Modify Default Write Concern

To modify a default write concern, change the “getLastErrorDefaults” setting in the replica set configuration.

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
rs.reconfig(cfg)
```

Include a timeout threshold for a write concern to prevent blocking write operations. Adding a timeout helps prevent the operation from blocking.

Read Preference

Read preferences define how MongoDB clients route read operations to replica set members. A client directs its read operations to the primary, which therefore contains the latest version of a document. You can distribute reads among the secondary members to improve the read performance or reduce latency for applications that do not require the up-to-date data.

The typical use cases for using non-primary read preferences are as follows:

- Running systems operations that do not affect the front-end application.
- Providing local reads for the geographically distributed applications.
- Maintaining availability during a failover.

Read Preference Modes

The read preference modes supported in a replica set are as follows:

Read Preference Modes	Description
Primary	• Default read preference mode • All operations read from the current replica set are primary
PrimaryPreferred	• Operations read from the primary member • When the primary is unavailable, operations read from a secondary member
Secondary	• All operations read from secondary members of the replica set
SecondaryPreferred	• Operations read from a secondary • Operations read from the primary when a secondary member is unavailable
Nearest	• Operations read from a member of the replica set with the least network latency

Blocking for Replication

The getLastError command in MongoDB performs up-to-date replication using the optional "w" parameter. The getLastError command given below is run to block until at least N number of servers have replicated the last write operation.

```
db.runCommand({getLastError: 1, w: N});
```

When specifying "w", the getLastError command takes an additional parameter, "wtimeout". This timeout allows the getLastError command to time out and return an error before the last operation has replicated to N servers.

Blocking the replication significantly slows down write operations, particularly for large values of "w".

Set "w" to two or three for important operations to yield a good combination of efficiency and safety.

Tag Set

Tag Sets allow tagging target read operations to select replica set members. Customized read preferences and write concerns assess tag sets as follows:

- Read preferences stresses on the tag value when selecting a replica set member to read from
- Write concerns ignore the tag value when selecting a member

You can specify tag sets with the following read preference modes:

- primaryPreferred
- secondary
- secondaryPreferred
- nearest

Tags are not compatible with the primary mode but compatible with the nearest mode. When combined together, the nearest mode selects the matching member, primary or secondary, with the lowest network latency.

Configure Tag Sets for Replica set

Tag sets allows customizing of write concerns and read preferences in a replica set. MongoDB stores tag sets in the replica set configuration object.

```
conf = rs.conf()
conf.members[0].tags = { "dc": "NYK", "rackNYK": "A"}
conf.members[1].tags = { "dc": "NYK", "rackNYK": "A"}
conf.members[2].tags = { "dc": "NYK", "rackNYK": "B" }
conf.members[3].tags = {"dc": "LON", "rackLON": "A"}
conf.members[4].tags = { "dc": "LON", "rackLON": "B"}
conf.settings = { getLastErrorModes : { MultipleDC : { "dc": 2}, multiRack: { rackNYK: 2 } }
rs.reconfig(conf)
```

Replica Set Deployment Strategies

A replica set architecture impacts the set's capability. You can use the following deployment strategies for a replica set.

Deploy an Odd Number of Member

For electing the primary member, add an arbiter if a replica set has an even number of members.

Consider Fault Tolerance

Adding a new member to a replica set may not increase fault tolerance. The additional members can provide support for some dedicated functions, such as backups or reporting.

Use Hidden and Delayed Members

Add hidden or delayed members to support dedicated functions, such as backup or reporting.

Load Balance on Read-Heavy Deployments

Distribute reads to secondary members on read-heavy deployments. Add or move members to alternate data centers and improve redundancy and availability.

Replica Set Deployment Strategies (contd.)

Some more Replica Set Deployment Strategies are given below:

Add Capacity Ahead of Demand	Add a new member to an existing replica set before new demands arise.
Distribute Members Geographically	Keep at least one member in an alternate data center as a backup in case of any data loss incidence. Set the priorities of these members to zero to prevent them from becoming primary.
Keep Majority in one Location	When electing the primary, all members must be able to see each other to create a majority. To enable the members elect the primary, ensure that most of the members are in one location.
Use Replica Set Tag Sets	This ensures that all operations are replicated at specific data centers. Using tag sets helps to route read operations to specific computers.
Use Journaling	Use journaling to safely write data on a disk in case of shutdowns, power failure, and other unexpected failures.

Replica Set Deployment Patterns

The common deployment patterns for a replica set are as follows:

Three Member Replica Sets

Minimum recommended architecture for a replica set

Four or More Member Replica Sets

Provides greater redundancy and supports greater distribution of read operations and dedicated functionality

Geographically Distributed Replica Sets

Members are distributed in multiple locations to protect data against facility-specific failures, such as power outages

Oplog File

The record of operations maintained by the master server is called the operation log or oplog. Each oplog document denotes a single operation performed on the master server and contains the following keys:

Timestamp (TS) for the Operation	Op Key	Namespace (NS)	O Key
An internal function to track operations. Contains a 4-byte timestamp and a 4-byte incrementing counter.	Type of operation performed as a 1-byte code, for example, “i” for an insert.	The collection name where the operation is performed.	Key for specifying the operation to perform. For an insert, this would be the document to insert.

Oplog stores only those operations that change the state of the database and is intended as a mechanism for keeping the data on slaves in sync with the master.

Replication State and Local Database

MongoDB maintains a local database called “local” to keep the information about the replication state and the list of master and slaves. The content of this database remains local to the master and slaves.

Slaves store the replication information in the local database. The unique slave identifier gets saved in the “me” collection and the list of masters gets saved in “sources” collection.

The timestamp stored in the “syncedTo” command is used as follows:

- **Master and Slave**: To understand how up-to-date a slave is.
- **Slave**: To query the oplog for new operations and find out if any operation is out of sync.

Replication Administration

To check the replication status, use the function given below when connected to the master.

```
configured oplog size: 10.48576MB
log length start to end: 34secs (0.01hrs)
oplog first event time: Tue Mar 30 2010 16:42:57 GMT-0400 (EDT)
oplog last event time: Tue Mar 30 2010 16:43:31 GMT-0400 (EDT)
now: Tue Mar 30 2010 16:43:37 GMT-0400 (EDT)
```

The oplog size and the date ranges of operations are contained in the oplog. In the given example, the oplog size is 10 megabyte and can accommodate about 30 seconds of operations.

The log length serves as a metric for servers that have been operational long enough for the oplog to “roll over.”

The functions given below will populate a list of sources for a slave, each displaying information such as how far behind it is from the master.

```
db.printSlaveReplicationInfo()
```

Demo—Check a Replica Set Status

This demo will show the steps to check the status of a replica set in MongoDB. Please refer to e-learning videos for this demo.

Sharding

Sharding is the process of distributing data across multiple servers for storage. The characteristics of sharding are as follows:

- Sharding adds more servers to a database and automatically balances data and load across various servers.
- Sharding provides additional write capacity by distributing the write load over a number of mongod instances.
- Sharding splits the data set and distributes them across multiple databases, or shards. Each shard serves as an independent database, and together, shards make a single logical database.
- Sharding reduces the number of operations each shard handles.

If a database has a 1 terabyte data set distributed amongst 4 shards, then each shard may hold only 256 GB of data. If the database contains 40 shards, then each shard will hold only 25 GB data.

When to Use Sharding?

Sharded clusters require a proper infrastructure setup, which increases the overall complexity of the deployment. Therefore, consider deploying sharded clusters only when your system shows the following characteristics:

- The data set outgrows the storage capacity of a single MongoDB instance.
- The size of the active working set exceeds the capacity of the maximum available RAM.
- A single MongoDB instance is unable to manage write operations.

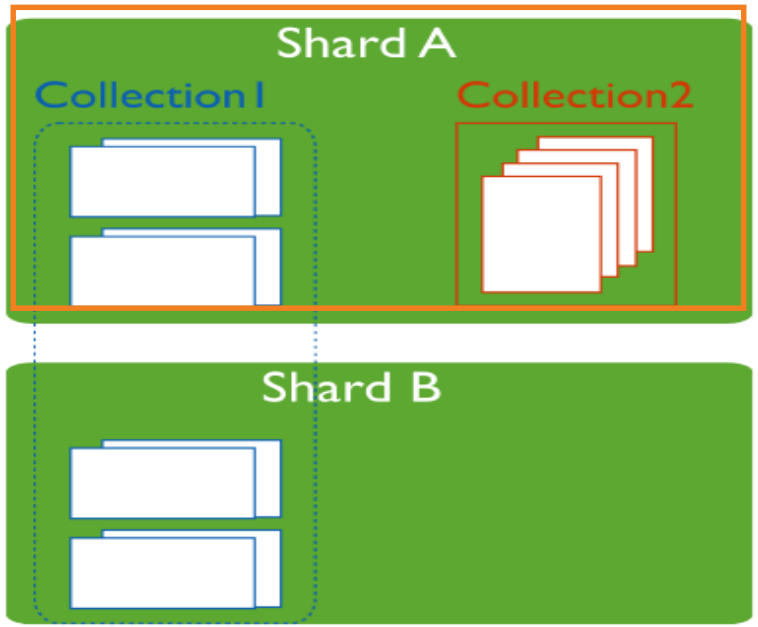
Deploy sharding if you expect that the read and write operations in your database are going to be increased in future.

What is a Shard?

A shard is a replica set or a single mongod instance that holds the data subset used in a sharded cluster. Each shard is a replica set that provides redundancy and high availability for the data it holds.

The characteristics of a shard are as follows:

- MongoDB shards data on a per collection basis.
- When directly connected to a shard, you will be able to view only a fraction of the data contained in a cluster.
- Data is not organized in any particular order in a shard.
- There is no guarantee that two contiguous data chunk will reside on any particular shard.

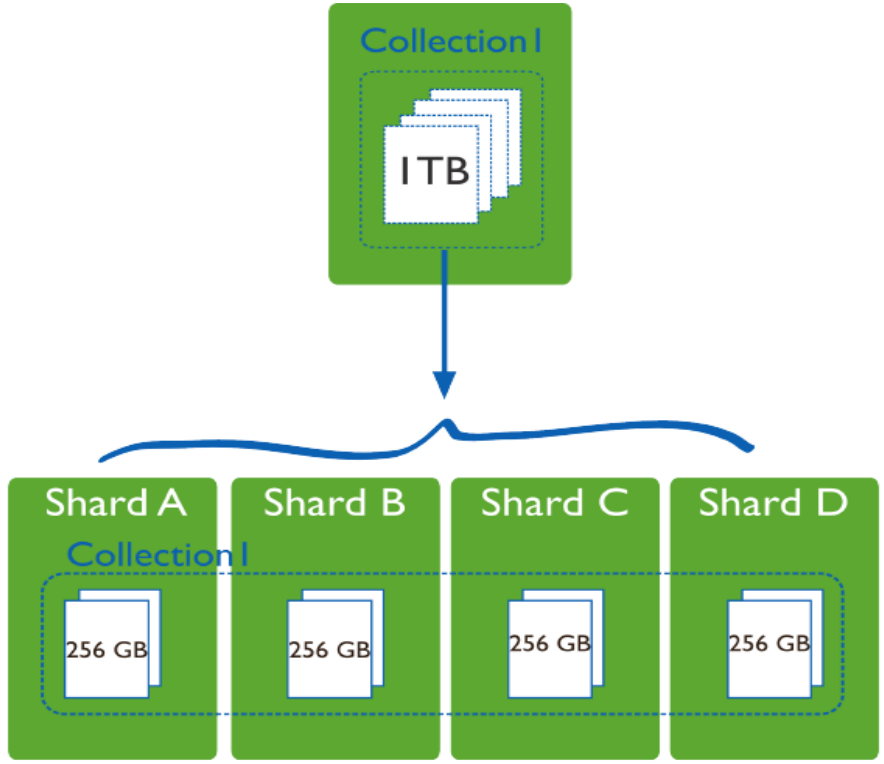


! Every database contains a “primary” shard that holds all the un-sharded collections in that database.

What is a Shard Key

When deploying sharding, you need to choose a key from a collection and split the data using the key's value. The characteristics of a shard key are as follows:

- Determines document distribution among the different shards in a cluster.
- Is a field that exists in every document in the collection and can be an indexed or indexed compound field.
- Performs data partitions in a collection.
- Helps distribute documents according to its range values.



Choosing a Shard Key

To enhance and optimize the performance, functioning and capability of your database, you need to choose the correct shard key.

Choose the appropriate shard key based on the following two factors:

- The schema of your data
- The way database applications query and perform write operations

Ideal Shard Key

An ideal shard key must have the following characteristics:

Must be Easily Divisible

An easily divisible shard key enables data distribution among shards. If shard keys contain limited number of possible values, then the chunks in shards cannot be split.

High Degree of Randomness

This ensures that a single shard distributes write operations among the cluster and does not become a bottleneck.

Target a Single Shard

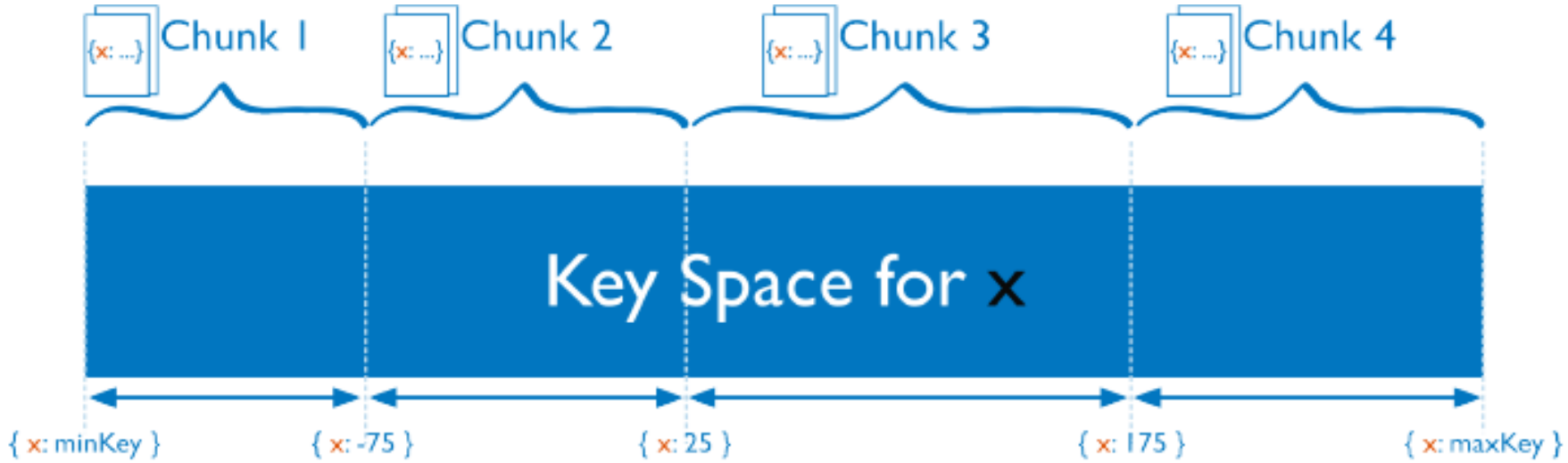
A shard key must target a single shard to enable the mongos program to return the query operations directly from a single mongod instance.

Use a Compound Shard Key

Compute a special purpose shard key or use a compound shard key if the existing field in a collection is not the ideal key.

Range-Based Shard Key

In range-based sharding, MongoDB divides data sets into different ranges based on the values of shard keys. In range-based sharding, documents having “close” shard key values reside in the same chunk and shard



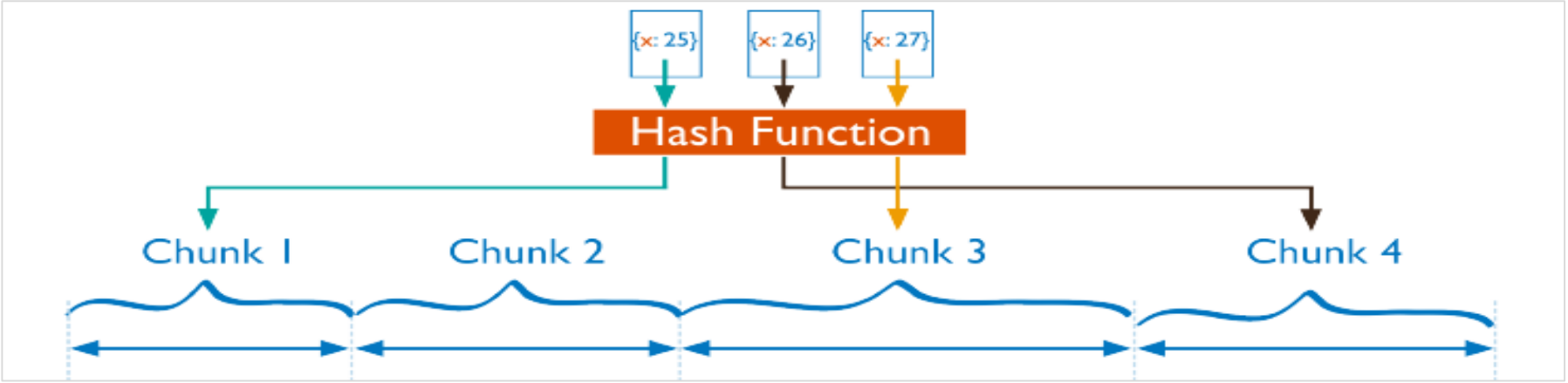
Range-based partitioning supports range queries because for a given range query of a shard key, the query router can easily find which shards contain those chunks.

Data distribution in range-based partitioning can be uneven, which may negate some benefits of sharding. For example, if a shard key field size increases linearly, such as time, then all requests for a given time range will map to the same chunk and shard. In such cases, a small set of shards may receive most of the requests and the system would fail to scale.

Hash-Based Sharding

For hash-based partitioning, MongoDB first calculates the hash of a field's value, and then creates chunks using those hashes. In hash-based partitioning, collections in a cluster are randomly distributed. In hash-based partitioning:

- Data is evenly distributed.
- Hashed key values randomly distribute data across chunks and shards.
- Range query on the shard key is ineffective.



Impact of Shard Keys on Cluster Operation

Some shard keys can scale write operations. A computed shard key with “randomness,” allows a cluster to scale write operations. To improve write scaling, MongoDB enables sharding a collection on a hashed index.

MongoDB improves write scaling using the following two methods:

Querying

A mongos instance enables applications to interact with sharded clusters. When mongos receives queries from client applications, it uses metadata from the config server and routes queries to the mongod instances. mongos makes querying operational in sharded environments.

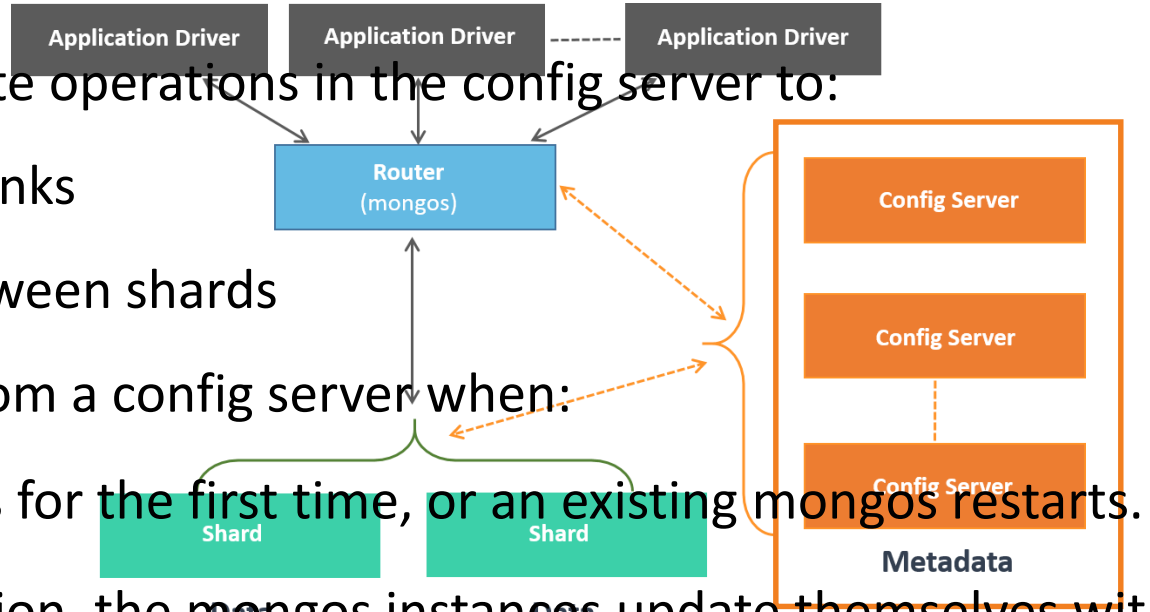
Query Isolation

Query execution will be fast and efficient if mongos can route to a single shard using a shard key and metadata stored from the config server. If your query contains the first component of a compound shard key then mongos can route a query to a single shard and thus provides good performance.

Production Cluster Architecture

Special mongod instances, such as config servers store metadata for a sharded cluster. Config servers provide consistency and reliability using a two-phase commit. Config servers do not run as replica sets and must be available to deploy a sharded cluster or to make changes to a cluster metadata.

Config servers store the metadata in the config databases. The mongos instance caches this data and uses it to route the reads and writes to shards.



MongoDB performs write operations in the config server to:

- Split the existing chunks
- Migrate a chunk between shards

MongoDB reads data from a config server when:

- A new mongos starts for the first time, or an existing mongos restarts.
- After a chunk migration, the mongos instances update themselves with the new cluster metadata.

Config Server Availability

Following are the characteristics of a config server:

- When one or more config servers are unavailable, the metadata of the cluster becomes read-only.
- Chunk migrations occur only when all the three servers are available.
- If the mongos instances are restarted before the config servers are available, the mongos become unable to route the reads and writes.
- Config servers must be available and intact throughout for clusters to become operable.

Query routers are the mongos processes that interface with the client applications and direct queries to the appropriate shard or shards. All the queries from client applications are processed by the query router. In the sharded cluster three query routers are recommended.

Production Cluster Deployment

A production cluster must have the following components:

Config Servers

Each of the three config servers must be hosted on separate machines. Each single sharded cluster must have an exclusive use of its config servers. Each cluster in a multiple sharded clusters must have a group of config servers.

Shards

A production cluster must have two or more replica sets or shards.

Query Routers (mongos)

A production cluster must have one or more mongos instances that act as the routers for the cluster. Configure the load balancer to enable a connection from a single client reach the same mongos.

Deploy a Sharded Cluster

To deploy a sharded cluster, perform the following sequence of tasks:

- **Step 1**: Create data directories for each of the three config server instances with the following command.

```
mkdir /data/configdb
```

- **Step 2**: Start each config server by issuing a command using the syntax given below.

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

- **Step 3**: To start a mongos instance, issue a command using the syntax given below.

```
mongo --host mongos0.example.net --port 27017
```

To start a mongos that connects to a config server instance running on the following hosts and on the default ports, issue the command given below.

Hosts
cfg0.example.net
cfg1.example.net
cfg2.example.net

```
sh.addShard( “mongodb0.example.net:27017")
```

Add Shards to a Cluster

To add shards to a cluster, perform the following steps.

- **Step 1:** From a mongo shell, connect to the mongos instance and issue a command using the syntax given below.

```
mongo --host <hostname of machine running mongos> --port <port mongos listens on>
```

If a mongos is accessible at node1.example.net on the port 27017 issue the command given below.

```
mongo --host node1.example.net --port 27017
```

- **Step 2:** Add each shard to the cluster using the sh.addShard() method given below.

```
sh.addSharding("<database>")
```

To add a shard for a replica set named rs1 with a member running on the port 27017 on mongodb0.example.net, issue the following command given below

```
sh.addShard( "rs1/node1.example.net:27017" )
```

To add a shard for a standalone mongod on the port 27017, issue the following command given below.

```
sh.addShard( "node1.example.net:27017" )
```

Demo—Create a Sharded Cluster

This demo will show the steps to create a sharded cluster in MongoDB.
Please refer to e-learning videos for this demo.

Enable Sharding for Database

Before you start sharding a collection, first enable sharding for the database of the collection.

To enable sharding, perform the following steps.

- **Step 1**: From a mongo shell, connect to the mongos instance and issue a command using the syntax given below.

```
mongo --host <hostname of machine running mongos> --port <port mongos listens on>
```

- **Step 2**: Issue the sh.enableSharding() method, specify the name of the database for which you want to enable sharding. Use the syntax given below:

```
sh.enableSharding("<database>")
```

Optionally, enable sharding for a database using the “enableSharding” command. For this, use the syntax given below.

```
db.runCommand( { enableSharding: <database> } )
```

Enable Sharding for Collection

To enable sharding for a collection, perform the following steps:

- **Step 1:** Determine the shard key value. The selected shard key value impacts the efficiency of sharding.
- **Step 2**: If the collection contains data, create an index on the shard key using the createIndex() method. If the collection is empty then MongoDB creates the index as a part of the sh.shardCollection().
- **Step 3:** To enable sharding for a collection, open the mongo shell and issue the sh.shardCollection() method using the syntax given below.

```
sh.shardCollection("<database>.<collection>", shard-key-pattern)
```

Enable Sharding for Collection (contd.)

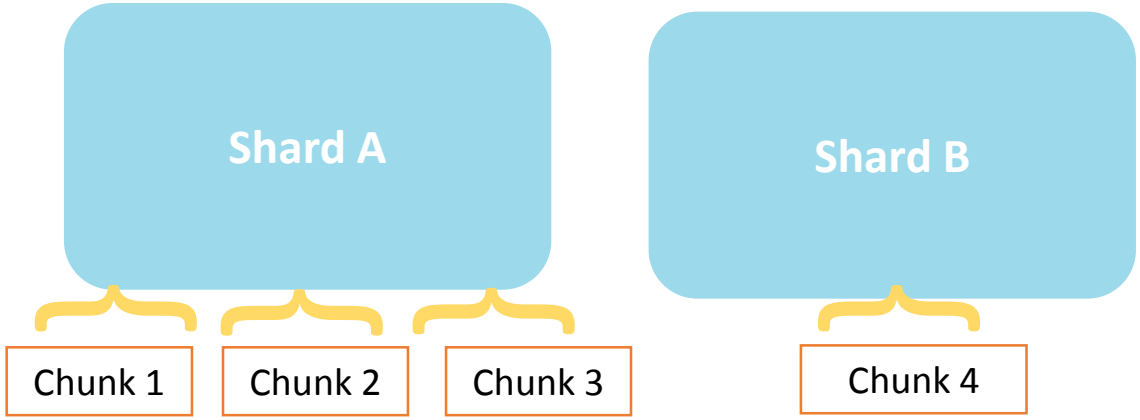
To enable sharding for a collection, replace the string <database>.<collection> with the full namespace of your database. This string consists of the name of your database, a dot, and the full name of the collection.

The example given below shows sharding collections based on the partition key.

```
sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )
sh.shardCollection("events.alerts", { "_id": "hashed" } )
```

Maintaining a Balanced Data Distribution

When you add new data or new servers to a cluster, it may impact the data distribution balances within the cluster. For example, some shards may contain more chunks than another shard or the chunk sizes may vary and some can be significantly larger than other chunks.

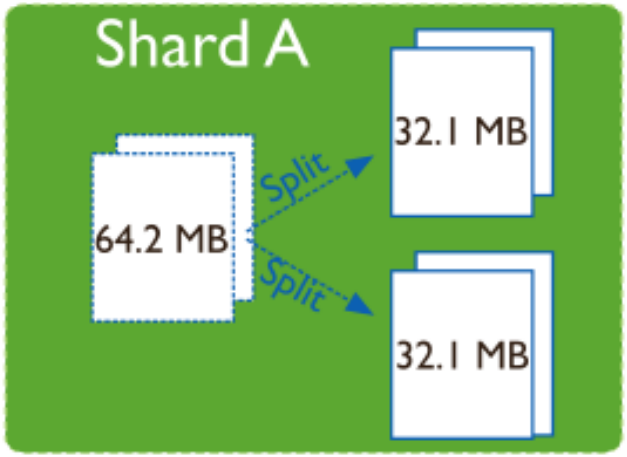


Splitting

Splitting is a background process that checks the chunk sizes. When a chunk exceeds a specified size, MongoDB splits the chunk into two.

When splitting a chunk, MongoDB:

- Uses the insert and update functions to trigger a split.
- Does not perform any data migration or affect the shards.



Chunk Size

In MongoDB, the default chunk size is 64 megabytes. You can alter a chunk size, which may impact the efficiency of the cluster.

Chunk size impacts migration in the following ways:

- Smaller chunks enable an even distribution of data but result in frequent migrations.
- Larger chunks encourage fewer migrations but leads to an uneven distribution of data.

When you split chunks, changing the chunk size affects with the following.

- Automatic splitting occurs only during inserts or updates. When you reduce the chunk size, all chunks may take time to split to the new size.
- Once you split chunks, they cannot be “undone”. When you increase the chunk size, the existing chunks must grow through inserts or updates until they reach the new size.

Special Chunk Type

Following are the special chunk types.

Jumbo Chunks

MongoDB does not migrate a chunk if :

- Size is more than the specified size
- The number of documents contained in the chunk exceeds the maximum number of documents per chunk to migrate

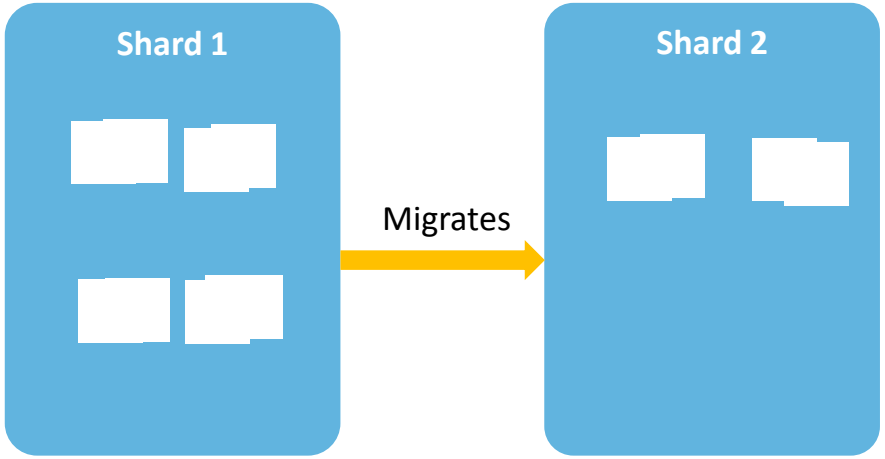
Indivisible Chunks

- Chunks can grow beyond the specified size but cannot be split
- Invisible chunks cannot be broken by the split process

Example: when a chunk represents a single shard key value.

Shard Balancing

MongoDB uses balancing to redistribute data within a sharded cluster. When the shard distribution in a cluster is uneven, the balancer migrates chunks from one shard to another to achieve a balance in chunk numbers per shard.



Chunk migrations is a background operation that occurs between two shards—an origin and a destination. The origin shard sends the destination shard to all the current documents. During the migration, if an error occurs, the balancer aborts the process and leaves the chunk unchanged in the origin shard.

Adding a new shard to a cluster may create an imbalance because the new shard has no chunks. Similarly, when a shard is being removed, the balancer migrates all the chunks from the shard to other shards. After all the data is migrated and the meta data is updated, the shard can be safely removed.

Shard Balancing (contd.)

Chunk migrations carry bandwidth and workload overheads, which may impact the database performance. Shard balancing minimizes the impact by:

- Moving only one chunk at a time
- Starting balancing only when two chunks reaches the migration threshold

You may want to disable the balancer temporarily for:

- MongoDB maintenance
- Prevent impacting the performance of MongoDB during peak load time

No Chunks	Migration Threshold
< 20	2
20-79	4
>80	8

Customized Data Distribution with Tag Aware Sharding

A MongoDB administrator can control sharding by using tags. Tags help control the balancer behaviour and chunks distribution in a cluster.

As an administrator you can create tags with the ranges of shard keys and assign them to the shards.

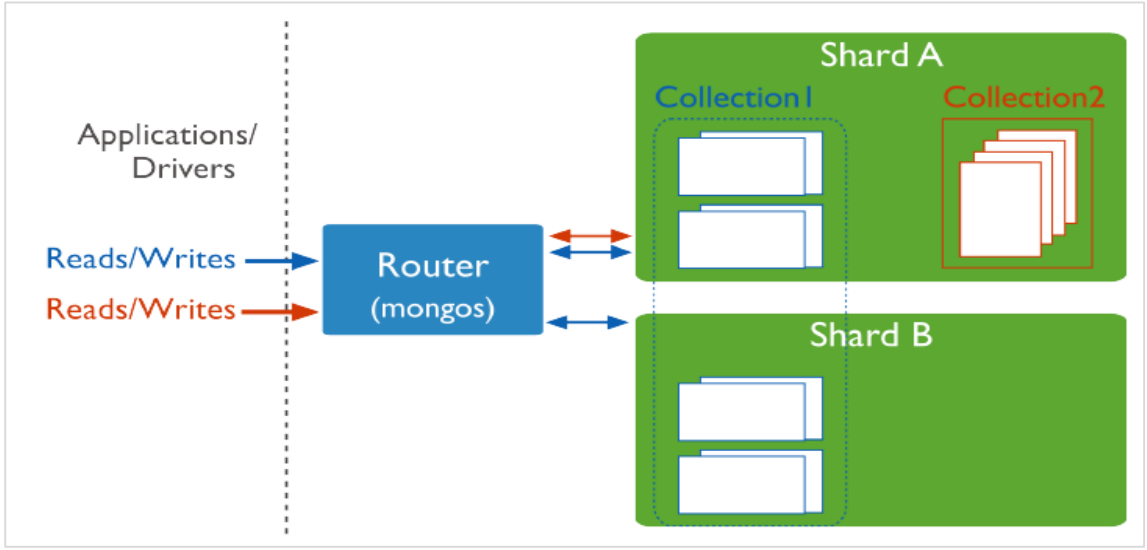
The balancer process migrates the tagged data to the shards that are assigned for those tags.

Tag Aware Sharding

In mongoDB, you can create tags for a range of shard keys to associate those ranges to a group of shards. Those shards receive all inserts within that tagged range.

The balancer that moves chunks from one shard to another obeys these tagged ranges.

The balancer moves or keeps a specific subset of the data on a specific set of shards and ensures that the most relevant data resides on the shard which is geographically closer to the client/application server.



Add Shard Tags

When connected to a mongos instance, use the sh.addShardTag() method to associate tags with a particular shard. The example given on the screen adds the tag NYC to two shards, and adds the tags SFO and NRT to a third shard.

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

To assign a tag to a range of shard keys, connect to the mongos instance and use the sh.addTagRange() method. The following operations assign:

- Two ranges of zip codes in Manhattan and Brooklyn, the NYC tag
- One range of zip codes in San Francisco, the SFO tag

```
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")
```

Remove Shard Tags

Shard tags exist in the shard's document in a collection of the config database. To return all shards with a specific tag, use the operations as given below.

```
use config
db.shards.find({ tags: "NYC" })
```

To return all shard key ranges tagged with NYC, use the following sequence of operations given below.

```
use config
db.tags.find({ tags: "NYC" })
```

The example given below removes the NYC tag assignment for the range of zip codes within Manhattan.

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

Quiz

QUIZ 1	What is the default size of a chunk?

a. 128 MB

b. 256 MB

c. 32 MB

d. 64 MB

QUIZ 1	What is the default size of a chunk?

a. 128 MB

b. 256 MB

c. 32 MB

d. 64 MB

The correct answer is **d**.

Explanation: 64 MB is the default chunk size.

QUIZ 2

Which of the following technique is used for scaling write operation in MongoDB?

a. Replication

b. Sharding

c. Indexing

d. Splitting

QUIZ 2

Which of the following technique is used for scaling write operation in MongoDB?

a. Replication

b. Sharding

c. Indexing

d. Splitting

The correct answer is **b**.

Explanation: Sharding is used for scaling write operations in MongoDB.

QUIZ 3	Which of the following replica set member does not store any data and just participates in the voting of primary?

a. Primary

b. Secondary

c. Hidden Member

d. Arbitar

QUIZ 3

Which of the following replica set member does not store any data and just participates in the voting of primary?

a. Primary

b. Secondary

c. Hidden Member

d. Arbiter

The correct answer is **d**.

Explanation: Arbiter does not store data and only participates in the voting process of primary

QUIZ 4

Which among the following statement is correct regarding replica set in MongoDB?

a. Write operation happens only at both primary and secondary machines

b. Write operation happens only at Secondary machines

c. Write operation happens only at primary machines

d. Write operation happens only at hidden machines

QUIZ 4

Which among the following statement is correct regarding replica set in MongoDB?

a. Write operation happens only at both primary and secondary machines

b. Write operation happens only at Secondary machines

c. Write operation happens only at primary machines

d. Write operation happens only at hidden machines

The correct answer is **c**.

Explanation: Write operation happens only at primary machines

Summary

Here is a quick recap of what was covered in this lesson:

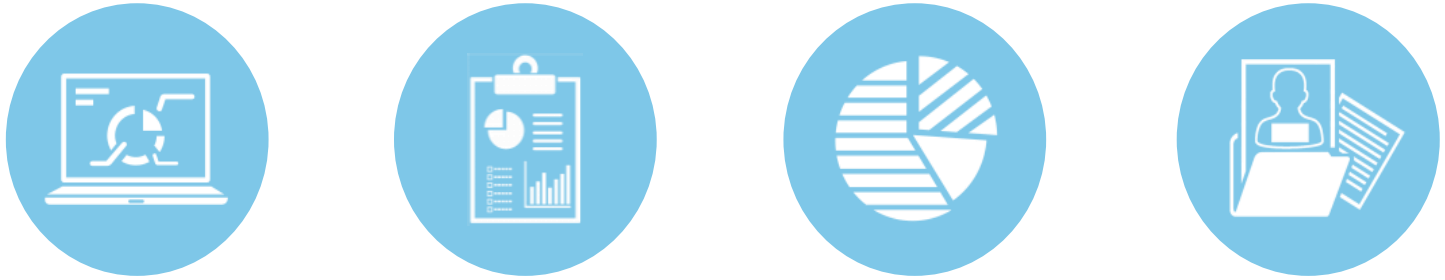
- Replication is recommended in the production setting to increase data availability and redundancy.
- Replica sets are recommended for new production deployments to replicate data in a cluster.
- A typical replica set in MongoDB consists of one primary and two secondaries.
- In addition to the primary and secondary, a replica set can have an arbiter, a priority zero member, a hidden member, and a delayed member.
- Consider deploying a sharded cluster when the data set exceeds the storage capacity of a single MongoDB instance, the size of the active working set exceeds the capacity of the maximum available RAM, or a single MongoDB instance is unable to manage the write operations.
- To deploy a sharded cluster, start the config server database instances and then start the three config server instances.
- Shard balancing is a background process that manages chunk migration for an even distribution of data shards in a cluster.

This concludes ‘Replication and Sharding.’

The next lesson is ‘Developing Java and Node JS Application with MongoDB.’

MongoDB Developer

Lesson 6: Developing Java and Node JS Application with MongoDB



After completing this lesson, you will be able to:

- Explain a capped collection
- Explain how Grid File System (GridFS) is used for storing large data
- Explain how to create a connection and perform the CRUD operation using Java Node JS applications
- Identify the steps to install the Java driver
- Explain the process to create a collection from a Java program
- Explain the procedure to use the Java program to perform functions, such as insert, update, and retrieve

Capped Collection

Capped collections in MongoDB are collections with predefined sizes and support operations that insert and retrieve documents. Capped collections function like circular buffers and exhibit the following behaviors:

- Preserve an insertion order and support higher insertion throughput.
- Ensure that the insertion order functions similar to the order on the disk or the natural order. This ensures that a document location on the disk remains unaltered.
- Delete the oldest documents in a collection automatically. They do not require scripts or explicit remove operations to do so.

Capped Collection Creation

Use the createCollection() method to create a capped collection.

When creating a capped collection, specify the maximum size of the collection in bytes. Use the syntax given below to specify the size:

```
db.createCollection( "serverlog", { capped: true, size: 100000 } )
```

If the specified size field is less than or equal to 4096 bytes, then the collection will have a cap of 4096 bytes. Otherwise, MongoDB will increase the size to make it an integer multiple of 256 bytes.

Use the max field given below to specify the maximum number of documents for the collection:

```
db.createCollection("server", { capped : true, size : 5242880, max : 5000 } )
```

Capped Collection Creation (contd.)

When creating a capped collection, you need to do the following:

- **Query a Capped Collection**: To retrieve documents in the reverse insertion order, issue the find() query along with the sort() method and set the $natural parameter to -1 as given below:

```
db.cappedCollection.find().sort( { $natural: -1 } )
```

- **Check if a Collection is Capped**: Use the isCapped() method as given in the command below to determine if a collection is capped:

```
db.collection.isCapped()
```

- **Convert a Collection to Capped**: Convert a non-capped collection to a capped collection with the convertToCapped command given below:

```
db.runCommand({"convertToCapped": "items", size: 100000});
```

Demo—Create a Capped Collection in MongoDB

This demo will show the steps to create a capped collection in MongoDB.
Please refer to e-learning videos for this demo.

Capped Collection Restriction

When updating a capped collection, you can make only in-place updates to documents. If the update operation increases the original document size, the operation does not succeed.

When updating a capped collection, remember the following:

- Create an index to avoid a table scan.
- When a document is updated to a smaller size, a secondary resyncs from the primary.
- The secondary replicates and allocates space based on the current size of the updated document.
- If the primary receives an update that restores the original document size, the primary accepts the update. The secondary fails and receives the error message “objects in a capped ns cannot grow”.

TTL Collection Features

Data is automatically removed from a capped collection once the collection grows beyond the threshold size and the number of documents defined. To allow flexibility for expiring data, consider MongoDB's time to live (TTL) indexes.

The features of TTL indexes are as follows:

- TTL indexes help delete data from normal collections using a special type field and a TTL value for the index.
- For TTL collections, mongod automatically removes data after a specified duration of time in seconds or at a specific clock time.

A special TTL index property supports the implementation of TTL collections as shown below:

```
db.log_events.createIndex( { "createdAt": 1 }, { expireAfterSeconds: 3600 }
)
```

Demo—Create TTL Indexes

This demo will show the steps to create TTL indexes in MongoDB. Please refer to e-learning videos for this demo.

GridFS

GridFS is a specification used for storing and retrieving files exceeding 16 MB.

GridFS does the following:

- Splits a file into different parts or chunks and stores each unit as a separate document.
- Stores files in two collections. In one collection, the file chunks are stored while the metadata is stored in the other collection.
- When returning a query result from GridFS, the driver or client reassembles the chunks as required.
- Allows range queries on the files stored in GridFS. For example, you can “skip” into the middle of a video or an audio file to access information.

GridFS Collection

You can store and retrieve files from GridFS using a MongoDB driver or the “mongofiles” command-line tool in the mongo shell. GridFS stores files in the following two collections:

- Fs.chunks store the binary chunks.
- FS.files store the file’s metadata.

Following are the characteristics of GridFS collections:

- Use a unique, compound index on the chunks collection for the files_id and n fields
- Number all chunks, starting with 0
- GridFS index allows efficient retrieval of chunks using the files_id and n values, as shown below:

```
cursor = db.fs.chunks.find({files_id: myFileID}).sort({n:1});
```

Issue the operation given below using the mongo shell if the driver does not create the earlier index:

```
db.fs.chunks.createIndex( { files_id: 1, n: 1 }, { unique: true } );
```

Demo—Create GridFS in MongoDB Java Application

simplilearn

This demo will show the steps to insert an image in MongoDB from a Java application.

Please refer to e-learning videos for this demo.

MongoDB Drivers and Client Libraries

Applications communicate with MongoDB through client libraries or drivers that handle all the interactions with the database in a language that is appropriate and sensible. The following languages are used for communication:

- JavaScript
- Python
- Ruby
- PHP
- Perl
- Java
- Scala
- C++ Haskell
- Erlang

To install the Java driver, perform the following steps:

- **Step 1**: Download the JAR file from the link given below.

https://github.com/mongodb/mongo-java-driver/downloads

- **Step 2.** Add the JAR file to your classpath. The Java classes you need to use in a normal application are in the com.mongodb and com.mongodb.gridfs packages.

The JAR file contains a number of other packages. You need these packages only to either modify the driver's internal functionality or extend its functionality.

Connecting to MonogDB from Java Program

To create a normal Java application, first create a new Java project in eclipse and add the MongoDB Java driver into its build and classpath. The Java driver of MongoDB provides the “com.mongodb.Mongo” class to connect to the MongoDB server. To establish a connection, create a new Mongo Object and pass the IP address and port number of the MongoDB server as a parameter as shown in the code given below.

```
import com.mongodb.MongoClient;
import com.mongodb.DB;
import com.mongodb.DBCollection;
class itemSearch {
public static void main(String[] args)
{
            MongoClient mongoClient = new MongoClient("localhost", 27017);
             DB db = mongoClient (“test");
                DBCollection items = db.getCollection(“items");
            }
    }
```

Create Collection From Java Program

You can create a collection from a Java program using the createCollection() method of the com.mongodb.DB class. A DB object instance is created from the return object of the mongoClient.getDB() method.

```
public static void main(String args[]) {
try {
// To connect to mongodb server
MongoClient mongoClient = new MongoClient("localhost", 27017);
// Now connect to your databases
DB db = mongoClient.getDB("test");
DBCollection coll = db.createCollection("mycol", null);
} catch (Exception e) {
System.err.println(e.getClass().getName() + ": " + e.getMessage());
}
}
}
```

Insert Documents From Java Program

Documents in Java must be instances of org.bson.DBObject. A document can be created in multiple ways in Java. However, the simplest way to create one is using the “com.mongodb.BasicDBObject” class. To create a document that is represented by the shell as item “Book”, “SoldQty": 500, use the command given below:

```
BasicDBObject doc = new BasicDBObject();
doc.put(“item”:“Book");
doc.put("SoldQty", 500);
```

Insert Documents Using Java Code Example

To insert a document in a collection, do the following:

- Create an instance of the DBCollection object by calling the getCollection() method of the DB object.
- Create BasicDBObject to represent the JSON document in the Java code.
- Call the append method to set the values for all the fields that need to be populated.
- Call the insert method of DBCollection after BasicDBObject is initialized.

Demo—Insert a Document Using Java

simplilearn

This demo will show the steps to insert a document in a collection using Java.

Please refer to e-learning videos for this demo.

Update Documents Using Java Code

The db.collection.update method is used to update an existing document in a collection.

In the code given on the screen, the DBCursor object is used to iterate through the list of documents in the while loop and update each document by inserting the field “like” with its value as 200.

```
public class UpdateDocuments {
public static void main(String args[]) {
try {
            DBCollection coll = db.getCollection("items");
            DBCursor cursor = coll.find();
      while (cursor.hasNext()) {
DBObject updateDocument = cursor.next();
updateDocument.put("likes", "200");
coll.update(null, updateDocument);
}
```

Demo—Retrieve Document Using Java

This demo will show the steps to retrieve a document from a collection using Java. Please refer to e-learning videos for this demo.

Delete Documents Using Java Code

To remove documents from a collection, use the DBCollection's remove method.

```
public class DeleteDocument {
public static void main(String args[]) {
try {
// To connect to mongodb server
DBCollection coll = db.getCollection("mycol");
System.out.println("Collection mycol selected successfully");
DBObject myDoc = coll.findOne();
coll.remove(myDoc);
DBCursor cursor = coll.find();
```

Demo—Delete Document Using Java

simplilearn

This demo will show the steps to delete a document from a collection using Java.

Please refer to e-learning videos for this demo.

Store Images Using GridFS API

simplilearn

To store images, use the code given below:

```
public class SaveImageApp {
        public static void main(String[] args) {
                try {           Mongo mongo = new Mongo("192.168.133.128", 27017);
                        DB db = mongo.getDB("imagedatabase1");
                        //DBCollection collection = db.getCollection("test");
                        String newFileName = "gridFS-java-image";
                        File imageFile = new File("c:\\butterfly.jpg");
                        // create a "photo" namespace
                        GridFS gfsPhoto = new GridFS(db, "photo");
                        GridFSInputFile gfsFile = gfsPhoto.createFile(imageFile);
                        gfsFile.setFilename(newFileName);
                        // save the image file into mongoDB
                        gfsFile.save();
```

Retrieve Images Using GridFS API

To retrieve DBCursor objects having reference to the files stored in GridFS, use the code given below:

```
DBCursor cursor = gfsPhoto.getFileList();
                        while (cursor.hasNext()) {
                                System.out.println(cursor.next());
                        }
                        // get image file by it's filename
                        GridFSDBFile imageForOutput =
gfsPhoto.findOne(newFileName);

                }
}
```

Remove Image Using GridFS API

To remove the image file form GridFS, use the code given below. You can use an instance of the GridFS object and call the remove method:

```
// remove the image file from mongoDB
                    gfsPhoto.remove(gfsPhoto.findOne(newFileName));



                    } catch (UnknownHostException e) {
                            e.printStackTrace();
                    } catch (MongoException e) {
                            e.printStackTrace();
                    } catch (IOException e) {
                            e.printStackTrace();
                    }

            }
}
```

Remove Image Using GridFS API

To work with a database, first create a connection. You can use MongoDB's native Node JS driver to create a connection with the MongoDB server. To install the MongoDB native drivers, first install the MongoDB module by using the npm command. Run the command given below.

```
npm install mongodb
```

npm is a package manager that provides a central repository for custom open source modules for Node JS and JavaScript.

npm helps manage modules, their versions and distribution easily. The npm install command is used to install the required module in a project.

Connection Creation Using Node JS

To create a connection using Node JS, perform the following steps.

- **Step 1**: Load the mongodb module using the code given below.

```
//lets require/import the mongodb native drivers.
  var mongodb = require('mongodb');
```

- **Step 2**: Define the URL you need to connect to.

```
// Connection URL. This is where your mongodb server is running.
var url = 'mongodb://localhost:27017/test';
```

- **Step 3**: Connect to the database: Use the MongoClient interface's connect method to connect to the database.

```
MongoClient.connect(url, function (err, db) {}
```

Insert Operations Using Node JS

After you have a db connection ready, you can communicate with the database and perform some basic operations on the MongoDB database.

When using the MongoDB native driver, you need to ensure that:

- The query function names and their parameters are similar to that of the native MongoDB commands.
- The query functions take the callback as the last argument. The callback function returns the first argument as an error and the second argument as a result. The result is actually the result or output provided by MongoDB on running these commands.

Insert Operations Using Node JS (contd.)

To create an items collection and save a few items in it, use the code given below.

```
// Get the documents collection
  var collection = db.collection('items');
            //Create some users
            var item1 = {item: 'Book', SoldQty: 500}
            var item2= {item: 'Pencil', SoldQty : 400}
            var item3= {item: 'Pen', SoldQty : 300}
            // Insert some items
            collection.insert([item1,item2,item3], function (err, result) {}
```

In the insert operation, you perform the following steps:

- Use the db.collection method
- Prepare data to be inserted
- Insert into the database

Demo—Perform CRUD Operation in Node JS

This demo will show the steps to perform CRUD operations in Node JS.
Please refer to e-learning videos for this demo.

Demo—Perform Insert and Retrieve Operations Using Node JS

simplilearn

This demo will show the steps to perform insert and retrieve operations in Node JS.
Please refer to e-learning videos for this demo.

Update Operations Using Node JS

simplilearn

You can update records using the collection.update method. The name and parameters of the update method are similar to that of native MongoDB queries. The code given below updates the documents from Node JS.

```
// Get the documents collection
        var collection = db.collection('items');
// Insert some users
collection.update({item: 'Pencil'}, {$set: {available: false}}, function (err, numUpdated) {
  if (err) {
    console.log(err);
  } else if (numUpdated) {
    console.log('Updated Successfully %d document(s).', numUpdated);
  } else {
    console.log('No document found with defined "find" criteria!');
 }
```

Retrieve Documents Using Node JS

To retrieve documents from MongoDB, trigger the find command on the collection object just as in the MongoDB shell. Use the example given below to retrieve a document.

```
// Get the documents collection
        var collection = db.collection('items');
        collection.find({item 'Pencil'}).toArray(function (err, result) {
if (err) {
  console.log(err);
} else if (result.length) {
console.log('Found:', result);
     } else {
       console.log('No document(s) found with defined "find" criteria!');
     }
```

Using DB Cursor to Retrieve Documents

You can use the db cursor instead of each document or use toArray to return the full array of all documents.

```
var collection = db.collection('items');
//We have a cursor now with our find criteria
   var cursor = collection.find({item: 'Book'});
            cursor.sort({item: -1});
            cursor.limit(10);
            cursor.skip(0);
   //Lets iterate on the result
   cursor.each(function (err, doc) {
     if (err) {
       console.log(err);
     } else {
       console.log('Fetched:', doc);
     }   });
```

Mongoose ODM Module in Node JS

MongooseJS is a popular library in NodeJS that provides a straightforward, schema-based solution to model your application data and includes built-in type casting, validation, query building, and business logic hooks.

Using the Mongoose interface features, such as automatic connection and connection pool management, you can easily work with MongoDB.

To work with mongoose, you need to first install the mongoose module. Run the npm install mongoose command to install the mongoose module.

Defining Schema Using Mongoose

In the code, you are doing the following things:

- Connecting to the database using the mongoose.connect.
- Creating a model using the mongoose.model method.

```
/Lets load the mongoose module in our program
var mongoose = require('mongoose');
//Lets connect to our database using the DB server URL.
mongoose.connect('mongodb://localhost/my_database_name');
/**
 * Lets define our Model for User entity. This model represents a collection in the
database.
 * We define the possible schema of User document and data types of each field.
 * */
var User = mongoose.model('User', {name: String, roles: Array, age: Number});
/* Lets Use our Models  * */
//Lets create a new user
var user1 = new User({name: 'modulus admin', age: 42, roles: ['admin', 'moderator',
'user']});
```

Defining Schema Using Mongoose (contd.)

You perform the following action using the code given below:

- **Create a new User Object**: You can create as many user objects using the model as desired, and modify them. Each object will represent a document in the database.
- **Save the user using the 'save' method**: The mongoose model has many methods available on the entity objects. In this case, you will use the save method to save the document in the database.

```
//Some modifications in user object
user1.name = user1.name.toUpperCase();
//Lets try to print and see it. You will see _id is assigned.
console.log(user1);
//Lets save it
user1.save(function (err, userObj) {
  if (err) {
    console.log(err);
  } else {
    console.log('saved successfully:', userObj);
  }
});
```

Demo—Use Mongoose to Define Schema

This demo will show the steps to define a schema and perform insert and query operations using mongoose in Node JS. Please refer to e-learning videos for this demo.

Demo—How to Run Node JS Using Mongoose

This demo will show the steps to run the Node JS application to perform insert and query operations using mongoose in Node JS. Please refer to e-learning videos for this demo.

Quiz

QUIZ 1	Which of the following syntax is used for creating a capped collection?

a. db.createCollection(“serverlog", { capped: **true**, size: 100000 })

b. db.cappedCollection(“serverlog", size: 100000)

c. db.createCollection({ capped: **true**, size: 100000 }, “serverlog")

d. db.cappedCollection(“serverlog", { capped: **true**, size: 100000 })

QUIZ 1	Which of the following syntax is used for creating a capped collection?

a. db.createCollection(“serverlog", { capped: **true**, size: 100000 })

b. db.cappedCollection(“serverlog", size: 100000)

c. db.createCollection({ capped: **true**, size: 100000 }, “serverlog")

d. db.cappedCollection(“serverlog", { capped: **true**, size: 100000 })

The correct answer is **a**.

Explanation: Option a is the correct syntax for creating a capped collection.

QUIZ 2	Which of the following technique is used for storing big size(more than 16 MB) file in MongoDB?

a. Capped collection

b. Sharded cluster

c. TTL

d. GridFS

QUIZ 2	Which of the following technique is used for storing big size(more than 16 MB) file in MongoDB?

a. Capped collection

b. Sharded Cluster

c. TTL

d. GridFS

The correct answer is **d**.

Explanation: GridFS is used to store more than 16MB data in MongoDB.

QUIZ 3	Which of the following technique is used for storing a file heavier than 16 MB in MongoDB?

a. collection.insert([item1,item2,item3]{}

b. collection.insert([item1,item2,item3], function (err, result) {}

c. GridFS.save();

d. DBCollection.insert(BasicDBObject)

QUIZ 3	Which of the following syntax is used for inserting a document using Java code?

a. collection.insert([item1,item2,item3]{}

b. collection.insert([item1,item2,item3], function (err, result) {}

c. GridFS.save();

d. DBCollection.insert(BasicDBObject)

The correct answer is **c**.

Explanation: GridFS is used for storing files heavier than 16 MB.

QUIZ 4	Which among the following statement is correct regarding mongoose in MongoDB?

a. It is a PHP library to connect with MongoDB

b. It is Java library to connect with MongoDB

c. It is used for modeling your application data in node js

d. It is python library to connect with MongoDB

QUIZ 4	Which among the following statement is correct regarding mongoose in mongoDB?

a. It is a PHP library to connect with MongoDB

b. It is Java library to connect with MongoDB

c. It is used for modeling your application data in Node JS.

d. It is python library to connect with MongoDB

The correct answer is **c**.

Explanation: It is used for modeling your application data.

Summary

Here is a quick recap of what was covered in this lesson:

- Capped collections in MongoDB contain collections with predefined sizes that support insert and retrieve operations.
- When creating a capped collection, you need to specify the size, query a collection, check if the collection is capped, and convert a collection to capped.
- The GridFS specification splits a file into different parts or chunks, and stores each unit as a separate document.
- To install the JAVA driver in MongoDB, you need to add the JAR file to your classpath.
- You can create a collection from a Java program using the createCollection() method of the com.mongodb.DB class.
- You can use the db cursor instead of each or use toArray to return the full array of all documents.
- You can use the native mongodb module and Mongo client to connect to MongoDB from Node JS.
- You can use the db.collection method to get the reference to your collection and then use insert, find, and delete methods.

This concludes ‘Developing Java and Node JS application with MongoDB.’