MongoDB

UNIT3-Part2

Overview

- MogoDB is a cross-platform, document oriented NoSQL database

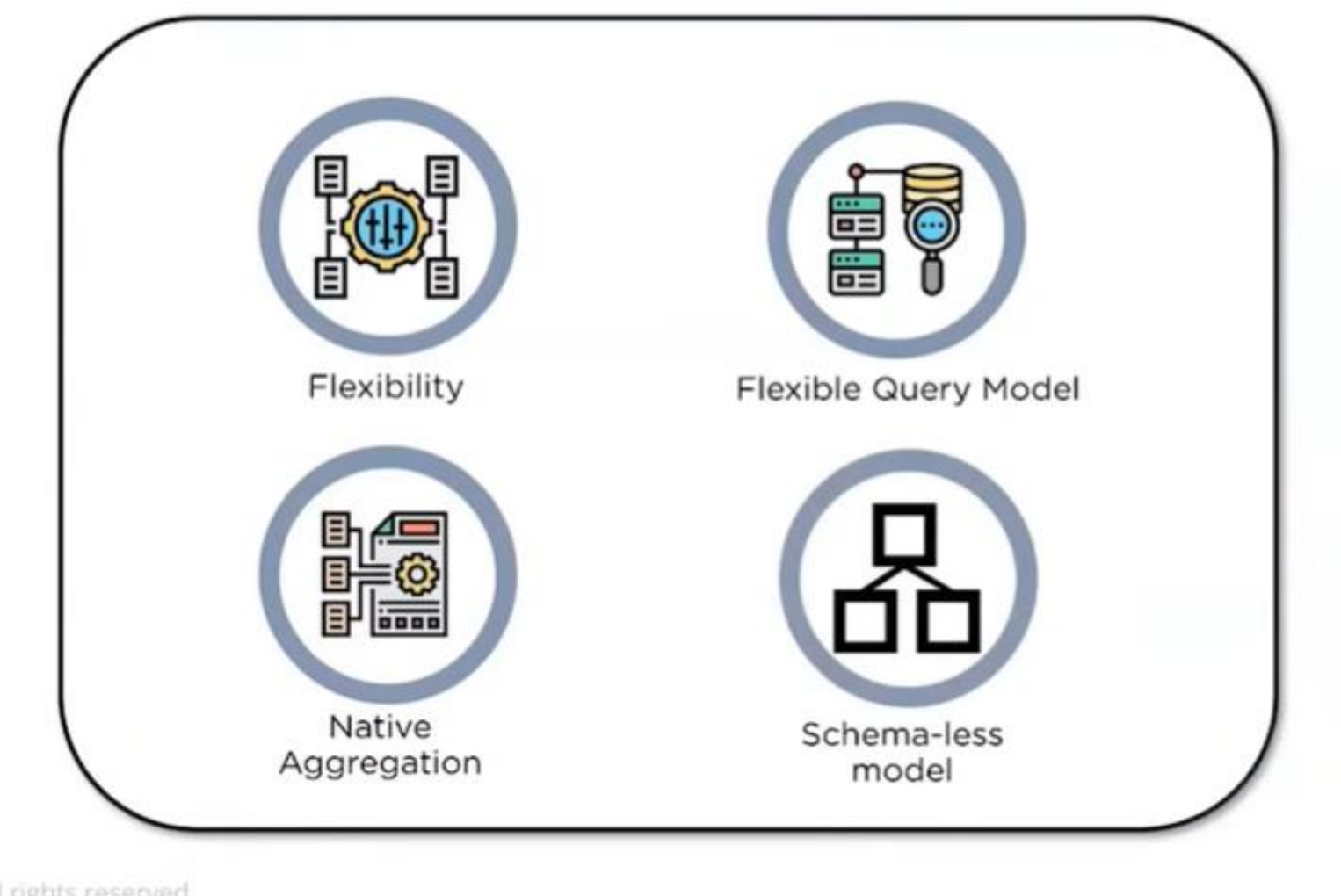
  Initially released on August 27,2009

  MongoDB replaces the concept of rows in RDBMS with that of documents

  It's a Schema Free model with High Performance, High Availability

  Easy Scalability

Why MongoDB



Flexibility

Nested and hierarchical structure of documents makes it is flexible and expressive

Flexible Query Model

Any part of the Database can be accessed by developing queries based on attribute values, regular expressions and ranges

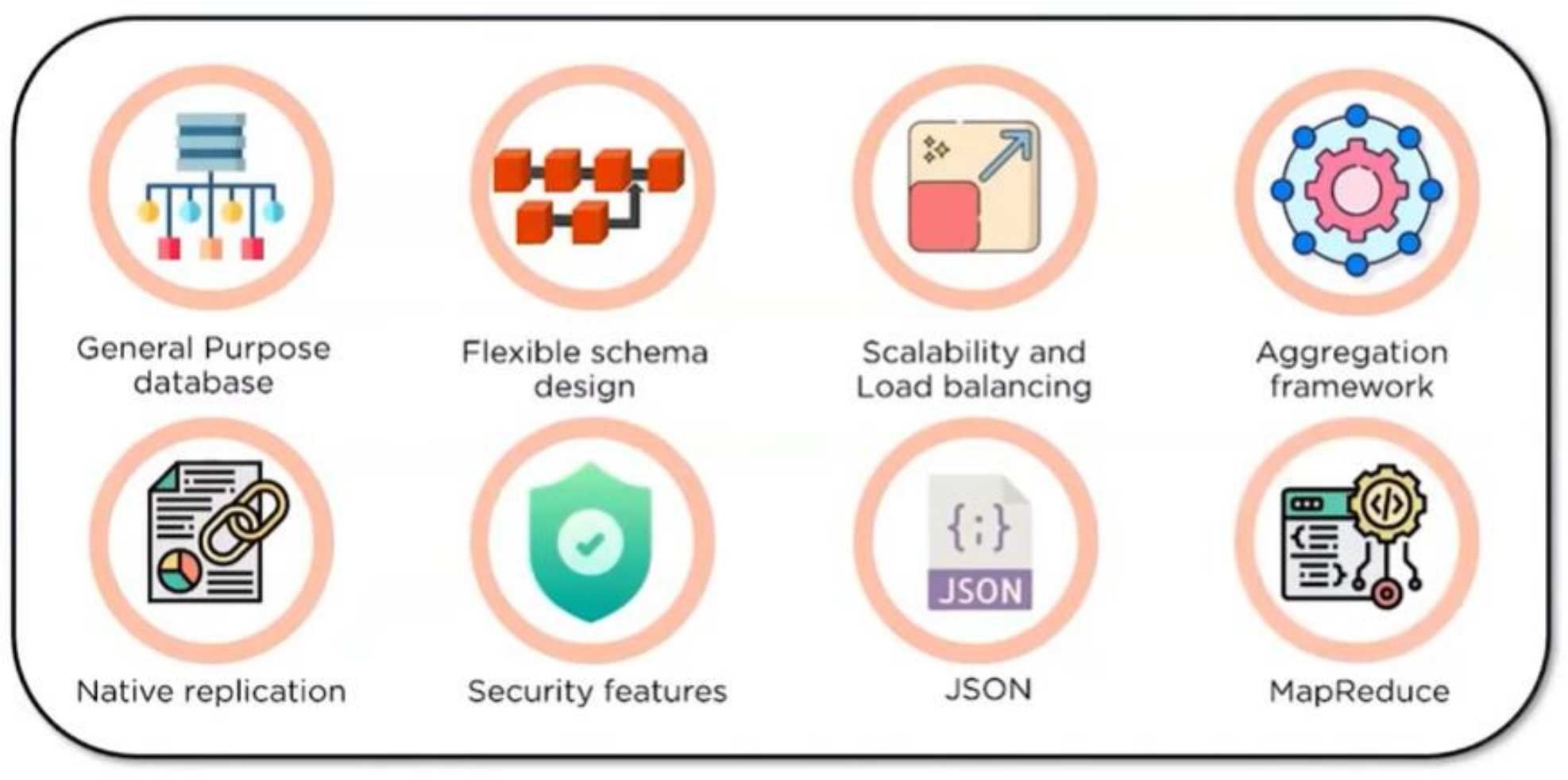
Navigate Aggregation

Allows the data to be loaded from and exported to any data source or application

Schema-less Model

Different properties can be represented in different ways

Characteristics of MongoDB



Characteristics of MongoDB

General Purpose Database

Can serve heterogeneous data sources and multiple purposes of an application

Flexible Schema

Non-defined attributes that can be modified on the fly makes it flexible

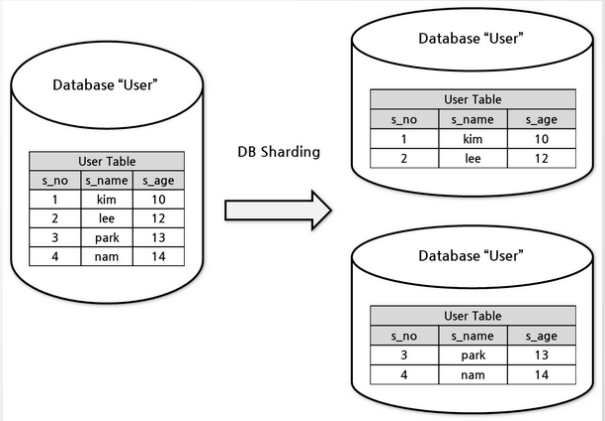
Scalability and Load-balancing

Mongodb scales both vertically and horizontally

Sharding concept achieves load-balancing

Aggregation Framework

Offers an ETL(Extract Load and Transform) Framework that eliminates the need for complex data pipeline



Characteristics of MongoDB

Native Replication

Data gets replicated across set of replications

Security Features

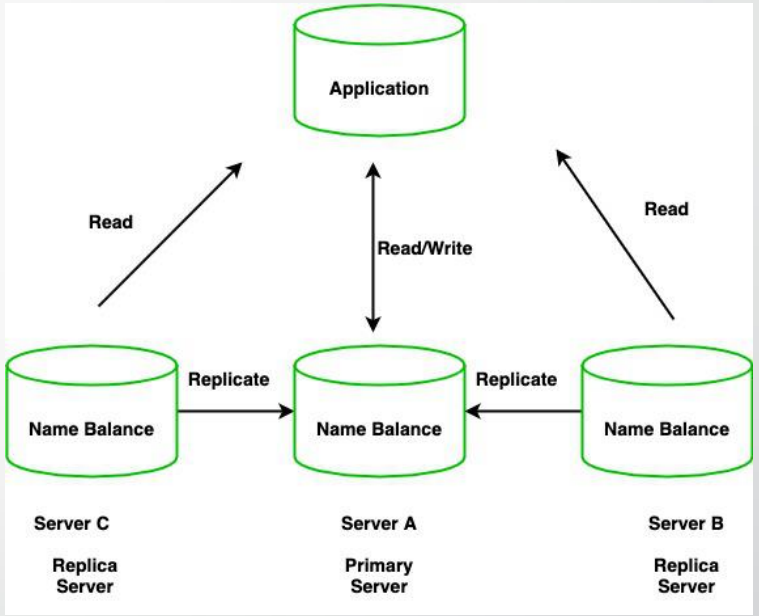
implements Authentication and Authorization

JSON compatibility

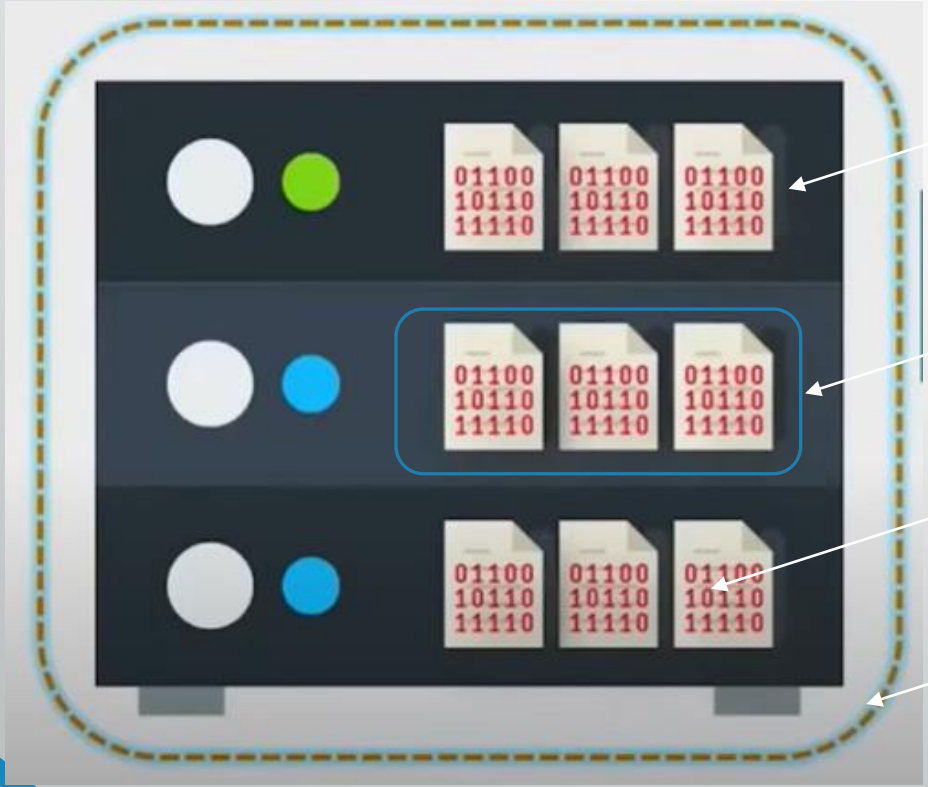
The database structure is compatible with the JSON structure which is being used by most of the Internet applications

Mapreduce

Works with Mapreduce readily



Working of MongoDB



MongoDB

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Relation between RDBMS and MongoDB

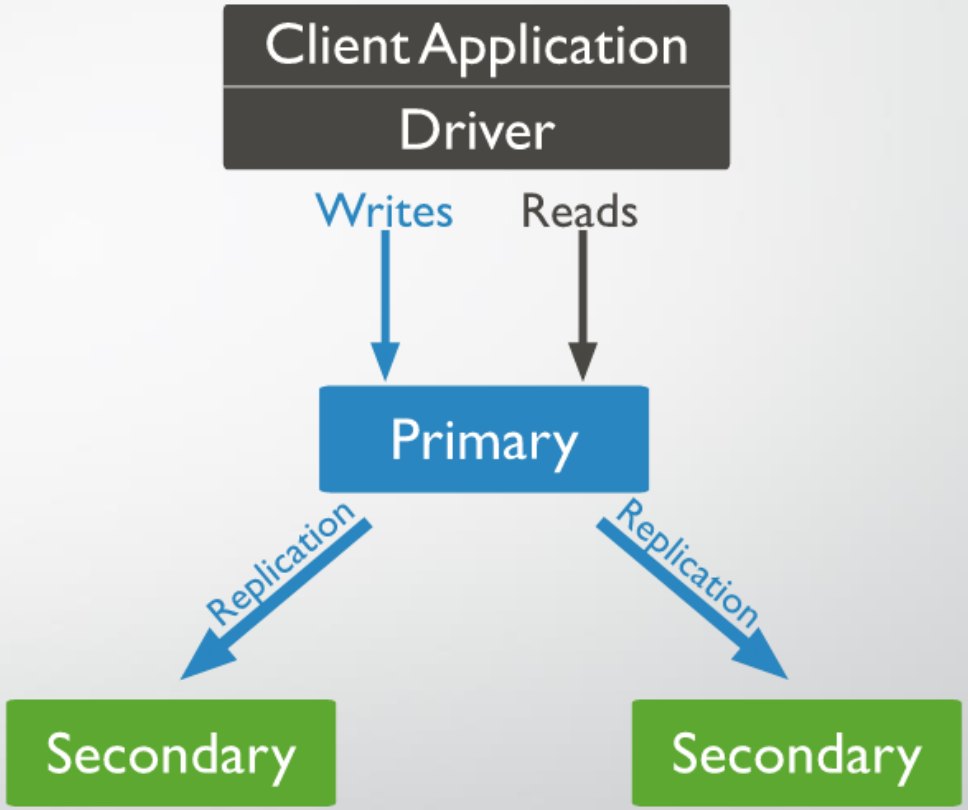
RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by MongoDB itself)
Database Server and Client	
mysqld/Oracle	mongod
mysql/sqlplus	mongo

Replica set in MongoDB

- A *replica set* in MongoDB is a group of mongod processes that maintain the same data set.
- Replica sets provide redundancy and high availability, and are the basis for all production deployments.
- Replication provides redundancy and increases data availability.
- With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.
- replication can provide increased read capacity as clients can send read operations to different servers.
- Maintaining copies of data in different data centers can increase data locality and availability for distributed applications.
- You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

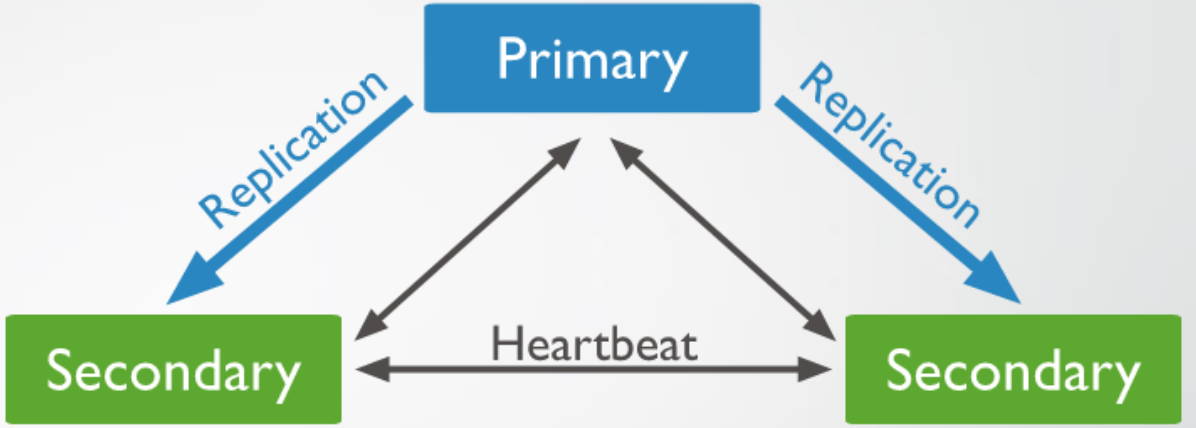
Replica set

- A replica set is a group of mongod instances that maintain the same data set
- A replica set contains several data bearing nodes and optionally one arbiter node.
- Of the data bearing nodes, one and only one member is deemed the primary node
- The other nodes are deemed secondary nodes.
- The primary node receives all write operations.
- The primary records all changes to its data sets in its operation log, i.e. oplog

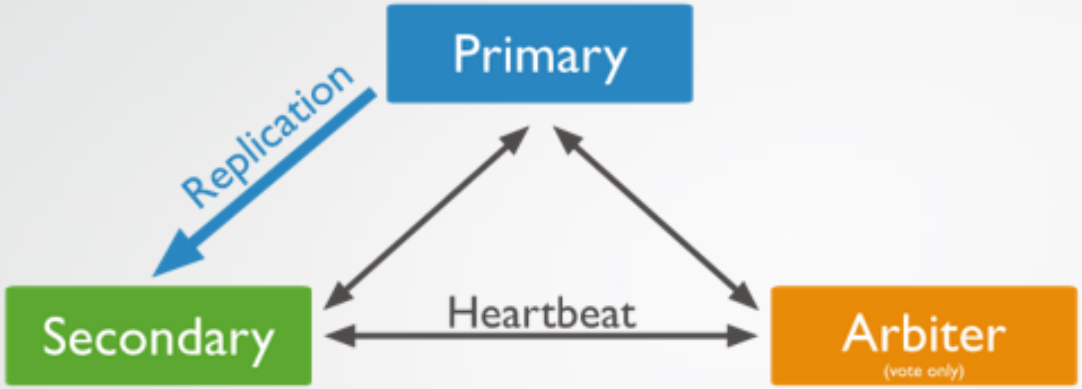


Replica set

- The secondaries replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set.
- If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.



Replica set



When we cannot afford for additional secondary node, we may choose to add a mongod instance to a replica set as an arbiter.

An arbiter participates in elections but does not hold data

An arbiter will always be an arbiter whereas

A primary may step down and become a secondary and

A secondary may become the primary during an election.

Sharding

Sharding is a method for distributing data across multiple machines.

A Database systems with large data sets or high throughput applications can challenge the capacity of a single server.

Ex: High query rates can exhaust the CPU capacity of the server

There are two methods for addressing system growth:
vertical and horizontal scaling.

MongoDB supports *horizontal scaling* through sharding.

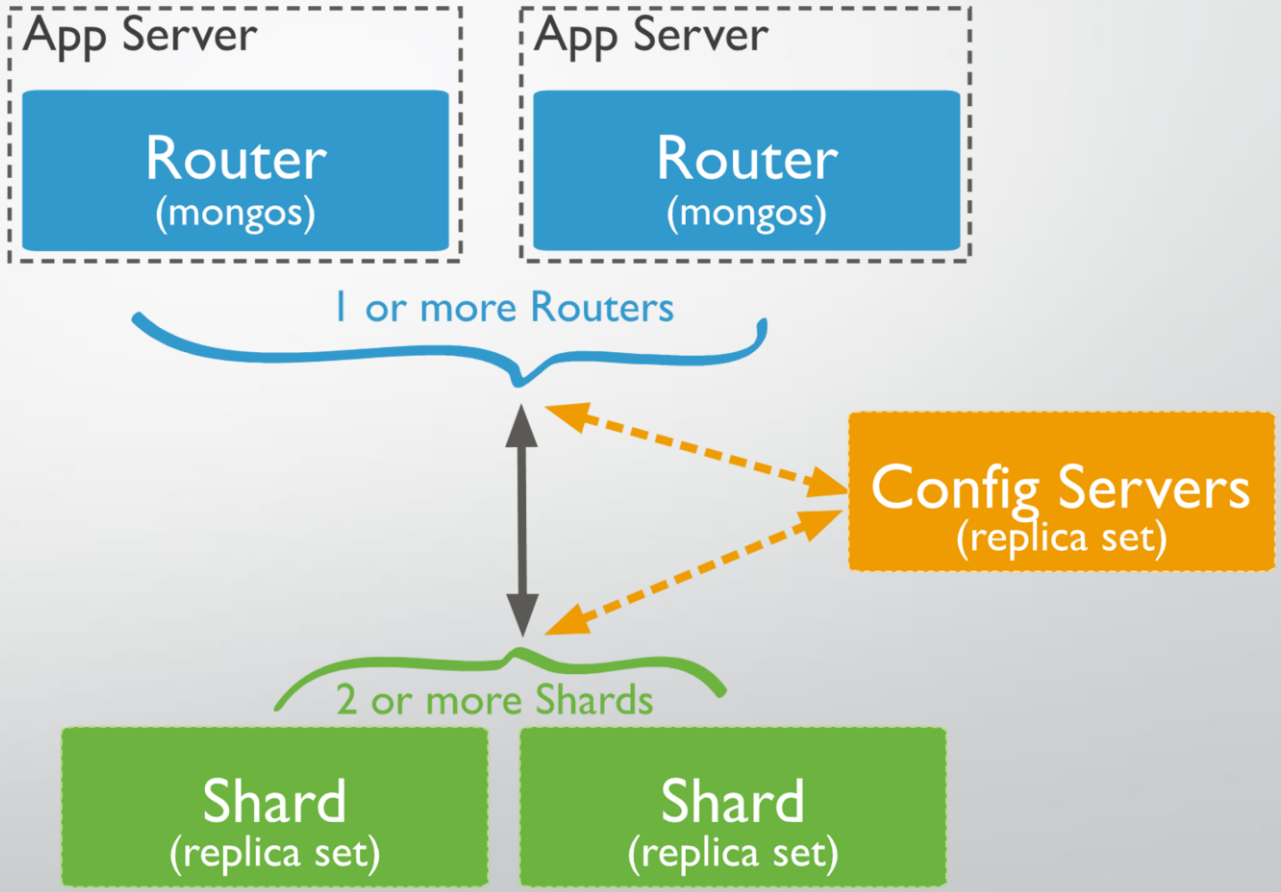
Sharded Cluster-Architecture

A MongoDB sharded cluster consists of the following components:

shard: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set.

mongos: The mongos acts as a query router, providing an interface between client applications and the sharded cluster.

config servers: Config servers store metadata and configuration settings for the cluster.



Shard Keys

MongoDB uses the shard key to distribute the collection's documents across shards.

The shard key consists of a field or multiple fields in the documents.

In version 4.2 and earlier, shard key fields must exist in every document for a sharded collection.
the choice of shard key cannot be changed after sharding.
a document's shard key field value is immutable.

Starting in version 4.4 , documents in sharded collections can be missing the shard key fields.
you can refine a shard key by adding a suffix field or fields to the existing shard key.

Missing shard key fields are treated as having null values when distributing the documents across shards but not when routing queries.

Missing Shard Keys

Document Missing Shard Key	Falls into Same Range As
{ x: "hello" }	{ x: "hello", y: null }
{ y: "goodbye" }	{ x: null, y: "goodbye" }
{ z: "oops" }	{ x: null, y: null }

To target documents with missing shard key fields, you can use the { $exists: false } filter condition on the shard key fields.

db.shardedcollection.find({ $or: [{ x: { $exists: false } }, { y: { $exists: false } }] })

For some write operations, which require an equality match on the shard key, we should include another filter condition with document id

{ _id: <value>, <shardkeyfield>: null }

Chunks

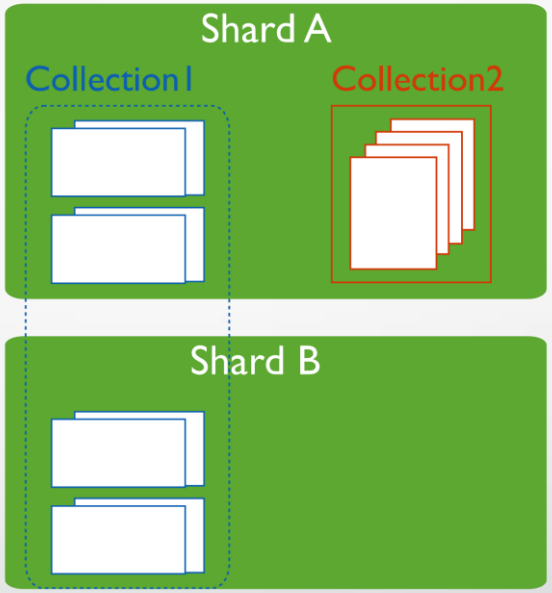
MongoDB partitions sharded data into chunks.

Each chunk has an inclusive lower and exclusive upper range based on the shard key

In an attempt to achieve an even distribution of chunks across all shards in the cluster, a balancer runs in the background to migrate chunks across the shards.

Sharded and Non-Sharded Collections

Sharded collections are partitioned and distributed across the shards in the cluster.



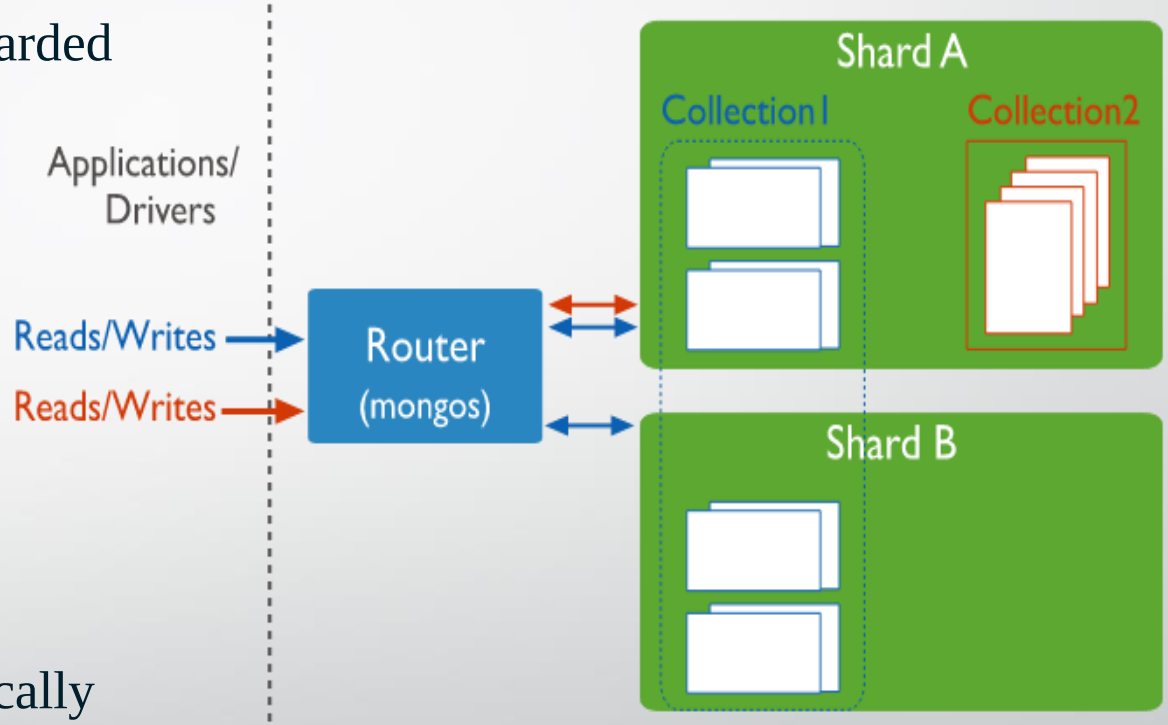
Unsharded collections are stored on a primary shard. Each database has its own primary shard.

Connecting to a Sharded Cluster

For a sharded cluster, the mongos instances provide the interface between the client applications and the sharded cluster.

The mongos instances route queries and write operations to the shards.

From the perspective of the application, a mongos instance behaves identically to any other MongoDB instance.



Sharding Strategy

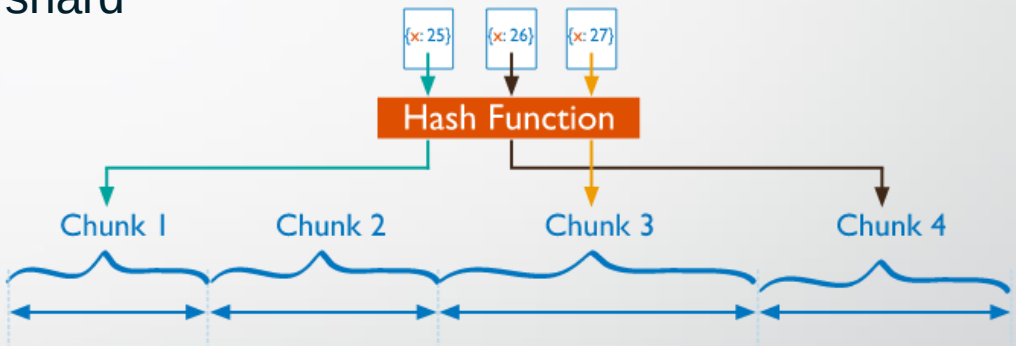
MongoDB supports two sharding strategies for distributing data across sharded clusters.

Hashed Sharding

Hashed Sharding involves computing a hash of the shard key field's value.

Each chunk is then assigned a range based on the hashed shard key values.

MongoDB automatically computes the hashes when resolving queries using hashed indexes.



While a range of shard keys may be "close", their hashed values are unlikely to be on the same chunk.

Data distribution based on hashed values facilitates more even data distribution, especially in data sets where the shard key changes monotonically

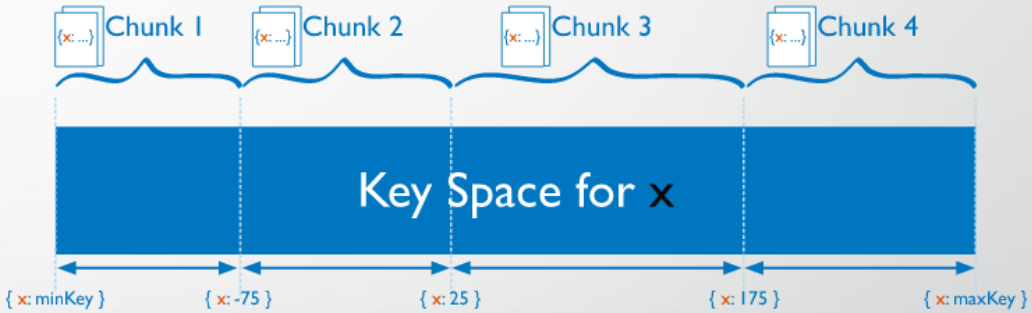
Sharding Strategy

Ranged Sharding

Ranged sharding involves dividing data into ranges based on the shard key values.

Each chunk is then assigned a range based on the shard key values.

A range of shard keys whose values are "close" are more likely to reside on the same chunk



The efficiency of ranged sharding depends on the shard key chosen. Poorly considered shard keys can result in uneven distribution of data, which can negate some benefits of sharding or can cause performance bottlenecks.

Environment

Installing MongoDB in ubuntu

Command to import the MongoDB public GPG(GNU Privacy Guard) key

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

Downloading MongoDB repository

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

update the repository

```
sudo apt-get update(upgrade)
```

install the MongoDB

```
apt-get install mongodb-10gen = 4.2
```

Environment

Start MongoDB

```
sudo service mongodb start
```

Stop MongoDB

```
sudo service mongodb stop
```

Restart MongoDB

```
sudo service mongodb restart
```

To use MongoDB run the following command.

```
mongo
```

Data Modelling

MongoDB's data will have flexible scheme.

- It is not mandatory for all the collections to have same set of fields.
- Common fields of different collections can hold different values of different datatype.

MongoDB provides two different Data models

Embedded Data Model

Organizes all the related documents in a single collection

Normalized Data Model

Organizes related documents separately by maintaining reference links

Embedded Data Model

```
{
    _id: ,

    Emp_ID: "10025AE336"

    Personal_details:
        {
         First_Name: "Parth",
         Last_Name: "Goel",
         Date_Of_Birth: "1985-09-26"
           },
    Contact:
     {
         email: "parth_goel@gmail.com",
         phone: "9876543210"
     },
     Address:
     {
         city: "Chicago",
        Area: "North America",
        State: "USA"
     }
}
```

Normalized Data Model

Employee:

```
{
 _id: <ObjectId101>,

Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
    _id: <ObjectId102>,
    empDocID: " ObjectId101",
    First_Name: "Parth",
    Last_Name: "Goel",
    Date_Of_Birth: "1985-09-26"
},
```

Contact:

```
{
   _id: <ObjectId103>,
   empDocID: " ObjectId101",
   email: "parth_goel@gmail.com",
   phone: "9876543210"
},
```

Address:

```
{
  _id: <ObjectId104>,
   empDocID: " ObjectId101",
   city: "Chicago",
   Area: "North America",
   State: "USA"
}
```

Datatypes

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your serve
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.

Datatypes

- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

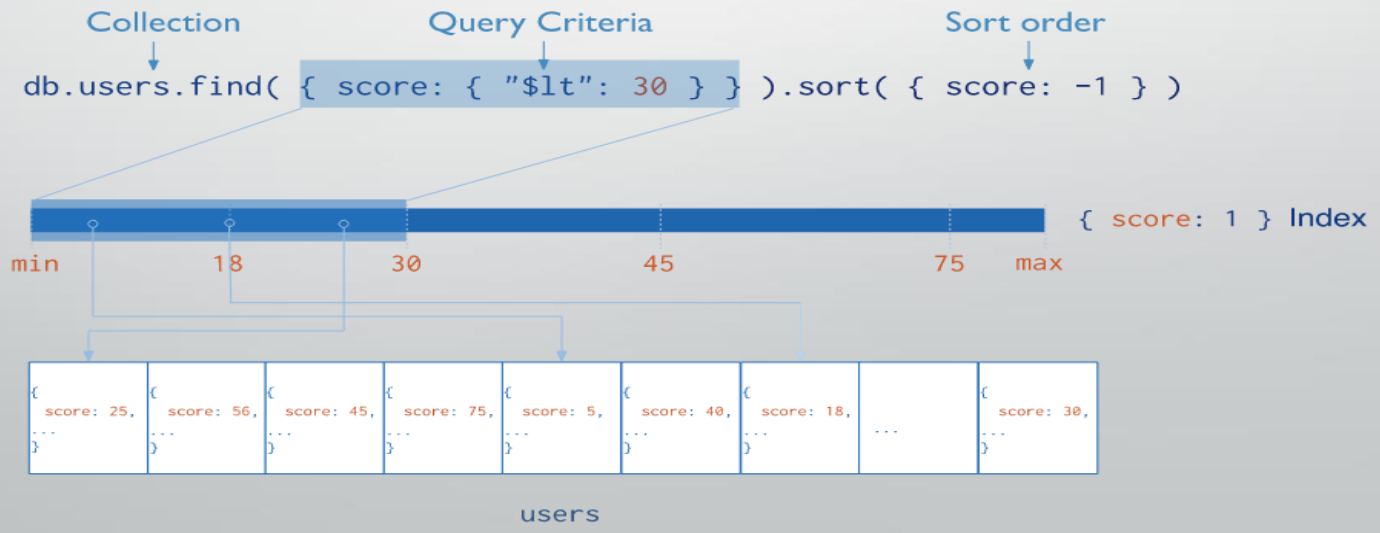
Indexing in MongoDB

Indexes support the efficient execution of queries in MongoDB

If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

The index stores the value of a specific field or set of fields, ordered by the value of the field



Indexing in MongoDB

- MongoDB creates a unique index on the _id field during the creation of a collection.

For creating an index in MongoDB we use

db.collection.createIndex()

For listing the indexes in MongoDB we use

db.<collection>.getIndexes()

To delete an index from a collection, use the **dropIndex** method while specifying the index name to be dropped.

db.<collection>.dropIndex(<Index Name / Field Name>)

Querying with Indexing

```
db.studentgrades.find({},{_id:0}).sort({subject:1, score:-1})
```

```
> db.studentgrades.find({},{_id:0}).sort({subject:1, score:-1})
{ "name" : "Tom", "subject" : "History", "score" : 65, "notes" : "Adequate" }
{ "name" : "Alex", "subject" : "Literature", "score" : 78 }
{ "name" : "Harry", "subject" : "Maths", "score" : 99, "notes" : "Exceptional Performance" }
{ "name" : "Barry", "subject" : "Maths", "score" : 92 }
{ "name" : "Kent", "subject" : "Physics", "score" : 87 }
>
```

Index Types

- **Single Field**
- **Compound Index**
- **Geospatial Index**
- **Text Search Indexes**

Single Field

```
db.collection.createIndex( { name: -1 } )
```

Compound Index

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

Geospatial Index

2dsphere

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

2d

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

where the <location field> is a field whose value is either a GeoJSON object or a legacy coordinates pair.

Text Search Indexes

```
db.reviews.createIndex( { comments: "text" } )
```

```
db.reviews.createIndex(
   {
     subject: "text",
     comments: "text"
   }
 )
```

MongoDB index properties

- Sparse index

```
db.studentgrades.createIndex({notes:1},{sparse: true})
```

- Partial index

```
db.studentgrades.createIndex(
{name:1},
{partialFilterExpression: {score: { $gte: 90}}}
)
```

- Unique index

```
db.studentgrades.createIndex({name:1},{unique: true})
```

Aggregation

Aggregation operations process multiple documents and return computed results.

They can be used to:

- Group values from multiple documents together.
- Perform operations on the grouped data to return a single result.
- Analyze data changes over time.

- Aggregation pipelines are the preferred method for performing aggregations.

  An aggregation pipeline consists of one or more stages that process documents:

  - Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
  - The documents that are output from a stage are passed to the next stage.
  - An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

Aggregate Stage Operators

- **$match:** It is used for filtering the documents can reduce the amount of documents that are given as input to the next stage.
- **$project:** It is used to select some specific fields from a collection.
- **$group:** It is used to group documents based on some value.
- **$sort:** It is used to sort the document that is rearranging them
- **$skip:** It is used to skip n number of documents and passes the remaining documents
- **$limit:** It is used to pass first n number of documents thus limiting them.
- **$unwind:** It is used to unwind documents that are using arrays i.e. it deconstructs an array field in the documents to return documents for each element.
- **$out:** It is used to write resulting documents to a new collection

Aggregate Accumulate Operators

- **sum:** It sums numeric values for the documents in each group
- **count:** It counts total numbers of documents
- **avg:** It calculates the average of all given values from all documents
- **min:** It gets the minimum value from all the documents
- **max:** It gets the maximum value from all the documents
- **first:** It gets the first document from the grouping
- **last:** It gets the last document from the grouping

Example

```
db.students.aggregate([{$match:{sec:"B"}},{$count:"Total student in sec:B"}])
```

In this example, for taking a count of the number of students in section B we first filter the documents using the **$match operator,** and then we use the **$count** accumulator to count the total number of documents that are passed after filtering from the $match.

Single Purpose Aggregation Methods

The single purpose aggregation methods aggregate documents from a single collection. The methods are simple but lack the capabilities of an aggregation pipeline.

Method	Description
db.collection.estimatedDocumentCount()	Returns an approximate count of the documents in a collection or a view.
db.collection.count()	Returns a count of the number of documents in a collection or a view.
db.collection.distinct()	Returns an array of documents that have distinct values for the specified field.