Intro to NoSQL

Sometimes we need RDBMS, sometimes we need NoSQL

- "When we build any toy app, we need RDBMS. When we need to scale, we need NoSQL"
  - That is not true always.

Difference between NoSQL and SQL

SQL

ID	Name	Add	Age	Role
12	Ram	23	30	~~SDE~~ Ass Prof

Add	city	country	Dis
23	Kolkata	Indra	—

Foreign key mapping

NOSQL

ID	value
12	{"Name" = "Ram", "address" = { id = 23, city = "kolkata", country = "India" }, "Age" = 30, Role = Ass Prof }

Here address is not object. Address is object under another object. We use the concept of nesting

⇓

JSON = Javascript Object Notation

Advantages

(1) **No need on Join**

In RDBMS, we need join, that is very expensive.
But in NOSQL, all the data are present in a block. So, it is very cheap

(2) **Flexible**

(a) In SQL, the distaict is NULL. But in NOSQL, if address is NULL, we don't care. Because, NOSQL does not care about scheme.

(b) Infact, in RDBMS, if we want to add a new attribute 'salary', We have to add a new column in SQL database, very expensive as we have some locks on the table and very risky to maintain consistency at this time.

But in NOSQL, We can easily add easily changable.

(3) **Build for scale (Horizontal scaling**

(4) **Built for aggregation**

Disadvantage

(1) If lots of updates, consistency is a problem
(ACID is not guaranted)
If ACID is not guaranted, we cannot have transaction in NoSQL database.

(2) NoSQL is not read optimize
We have to read entire data to find age. In SQL, we can directly go to the particular column. So, SQL is read optimized.
In NoSQL, read times are slower.

(3) We cannot have foreign key constrain

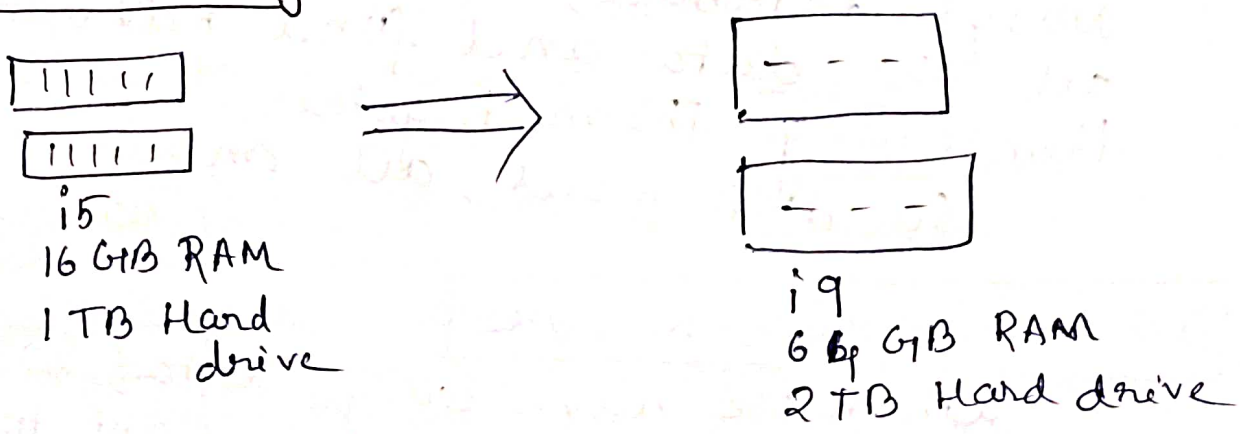
(4) Joins are hard. Join two blocks in NoSQL database, we have to go through all the data and find relevant attribu then merge them together.
So, it is hard, all manual.

When we use NoSQL

When we have large data, we want to keep them together. We want to make fewer updates, NoSQL is better.
YouTube still don't use NoSQL database

In SQL we use normalization to not waste memory.
1NF, 2NF, 3NF, BCNF.
There is no duplication.
SQL to fetch data
SQL is first choice for any database.
Now we have huge data, not only text, but also audio, vedio files, unstructured data, data generated by users, data generated by machines, IoT devices, which generates lot of data.
We have to store the data nowadays.
So the concept comes, we can store data, which is not structured (scheme)
We don't know how many attributes, how many columns. But nowadays, we don't know what are the data we are getting.
Scalability is another issue.

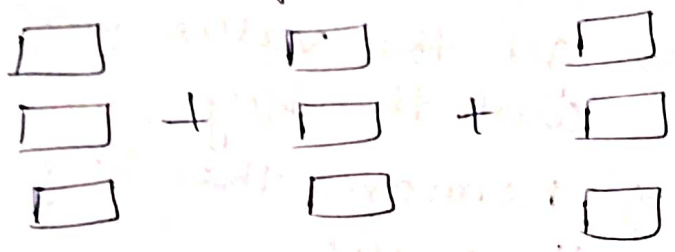
Vertical scaling



But there is a limit for CPU

Horizontal scaling

Not dependent of one server, we can have multiple server.



Key : value

"name" : "Ravi"
"Subject" : "Database"
"Address" : { "city" : "Kolkata",
"country" : "India",
}

- This is the simplest NoSQL database
- Every single item in the database is stored as an attribute name (or 'key') together with its value

Key 1 → value 1
Key 2 → value 2

- Data is stored as a collection of key/value pairs.
- The value can be integer, string or complex data types such as set of data.
- The key in a key-value pair must be unique.
- Data is retrieved via an exact match on key.

3 operations performed on a key-value database are:

(1) put (key, value) ⇒ insert or update
(2) get (key) ⇒ get the value associated with the key.
(3) delete (key) ⇒ remove the key and its value

Key value database

- Redis
- Riak
- Oracle NoSQL

Key-value database scheme

Student relation

Sid	Sfname	SPhone
16	ABC	9567
18	XYZ	9781

The above student relation is represented in key-value database schema as

Student : 16 : sfname (key) = "ABC"
Student : 16 : sphone = 9567
Student : 18 : sfname = "XYZ"
Student : 18 : sphone = 9781

table_name : primary-key-value : attribute_name (key) = value

Document-oriented database

These are similar to key-value databases in that, there is a key and a value
- But in a document database, the value contains structured or semi-structured data. The structured / semi-structured value is referred to as document.

The document can be stored in JSON format or XML format.

eg MongoDB
CouchDB
DocumentDB

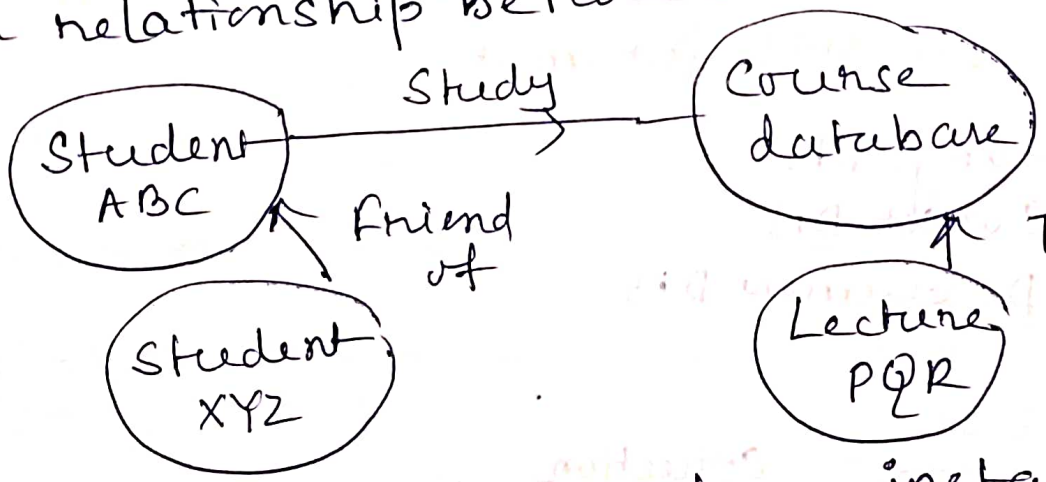

```
{ _id : 16,
  sfname : "ABC",
  sphone : 9657
}

{ _id : 18
  sfname : "XYZ"
  sphone : "9781
}
```

Enclose document with a pair of curly braces.

Graph database

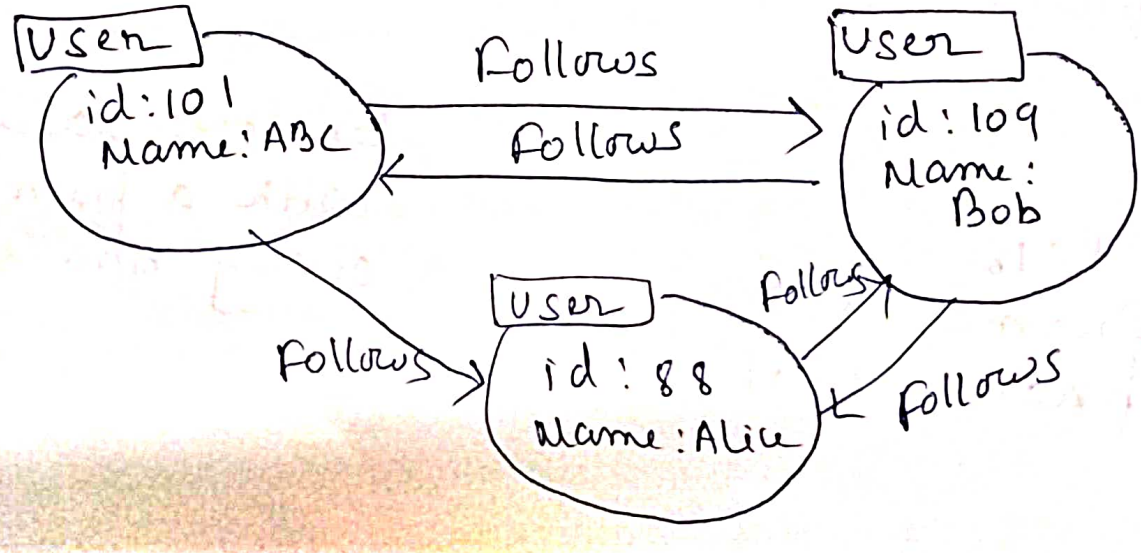
- It organizes data in the form of a graph
- A graph database contains a collection of nodes and edges.
- A node represents an entity, and an ~~he~~ ~~ong~~ edge represents the connection or relationship between two entities.



A node student, whose instance is ABC

eg Neo4J
OrientDB
ArangoDB

eg Graph database model of users in a social media network.



Columnar database

Data is stored in columns rather than rows. It speed up the time required to return a particular query.

eg Apache Cassandra, Amazon Redshift

id number	Last name	Salary
82	ABC	7000
23	XYZ	8000
34	PQR	9000

in columnar database

82, 23, 34; ABC, XYZ, PQR; 7000, 8000, 9000.

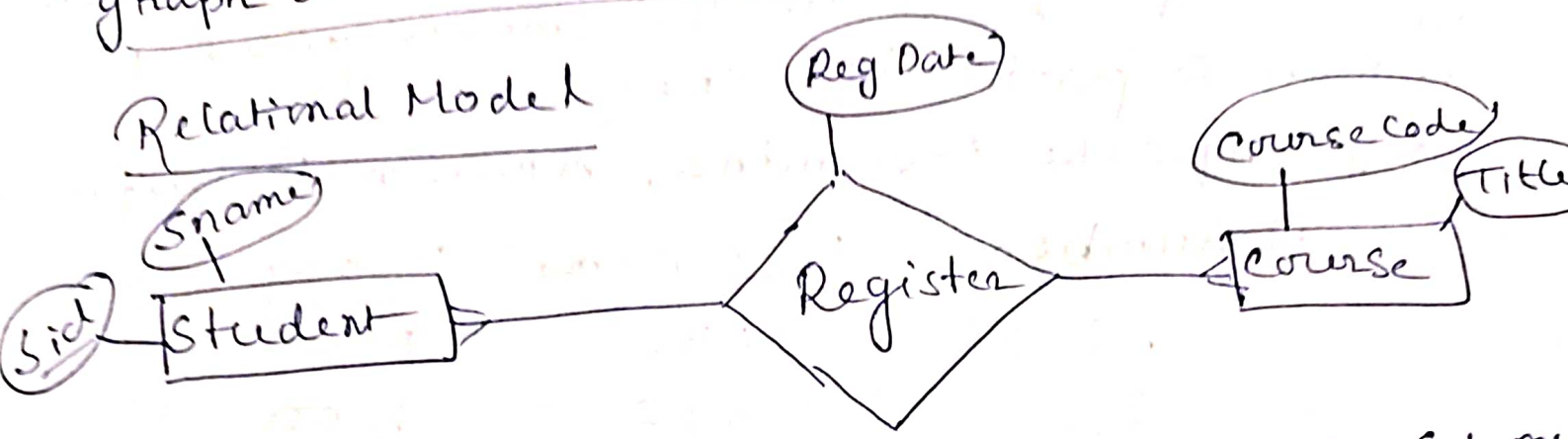
in row oriented database

82, ABC, 7000; 23, XYZ, 8000; 34, PQR, 9000.

It delivers high performance on aggregation queries like SUM, COUNT, AVG, MIN, etc as data are ~~read~~ available in a column.

Comparison of relational, document and graph database models

Relational Model



Student (Sid, SName)
Course (Course code, Title)
Register (Sid, Course code, Reg Date)

Scheme

Student Table

Sid	Sname
16	ABC

Course Table

Course code	Title
CS01	Intro to DB
CS02	DE

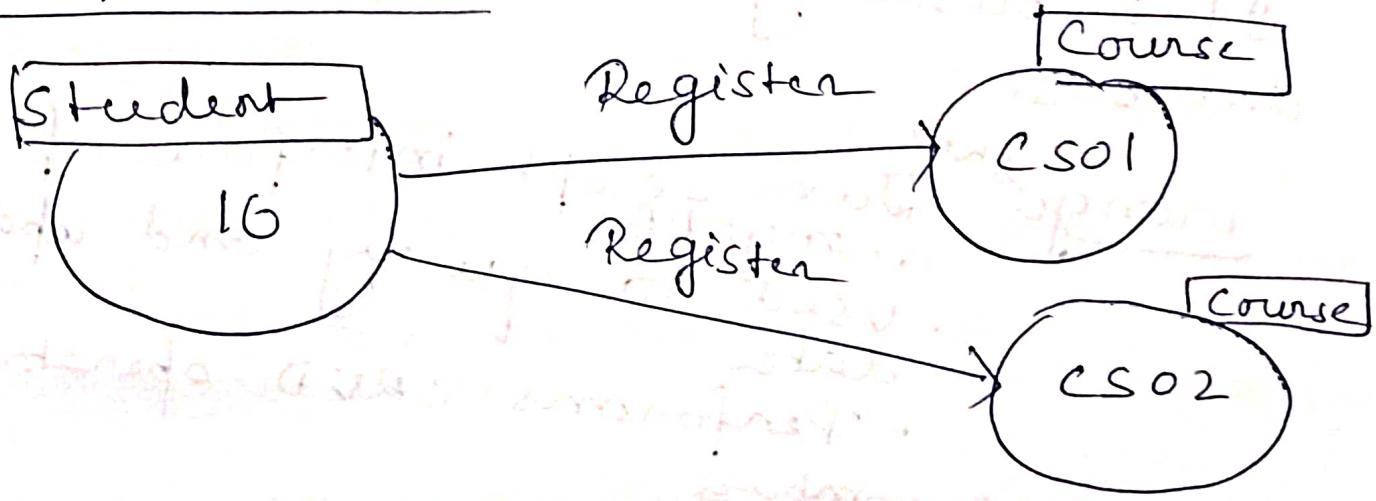
Register Table

Sid	Course code	Reg Date
16	CS01	2023-01-01
16	CS02	2023-01-01

document model

```
{ sid: "16",
  SName: "ABC",
  Course: [{ course code: "CS01",
             Title: "Intro to DB",
             RegDate "2023-01-01",
           },
           { Course code: "CS02",
             Title: "DE",
             RegDate: "2023-01-01",
           }]
}
```

Graph model



What is MongoDB

- MongoDB is a document database designed for ease of development and scaling.
- Easy to use NoSQL database
- Available as community and enterprise edition
- The community ~~edtg~~ edition itself is powerful.

Mongo vs MongoD

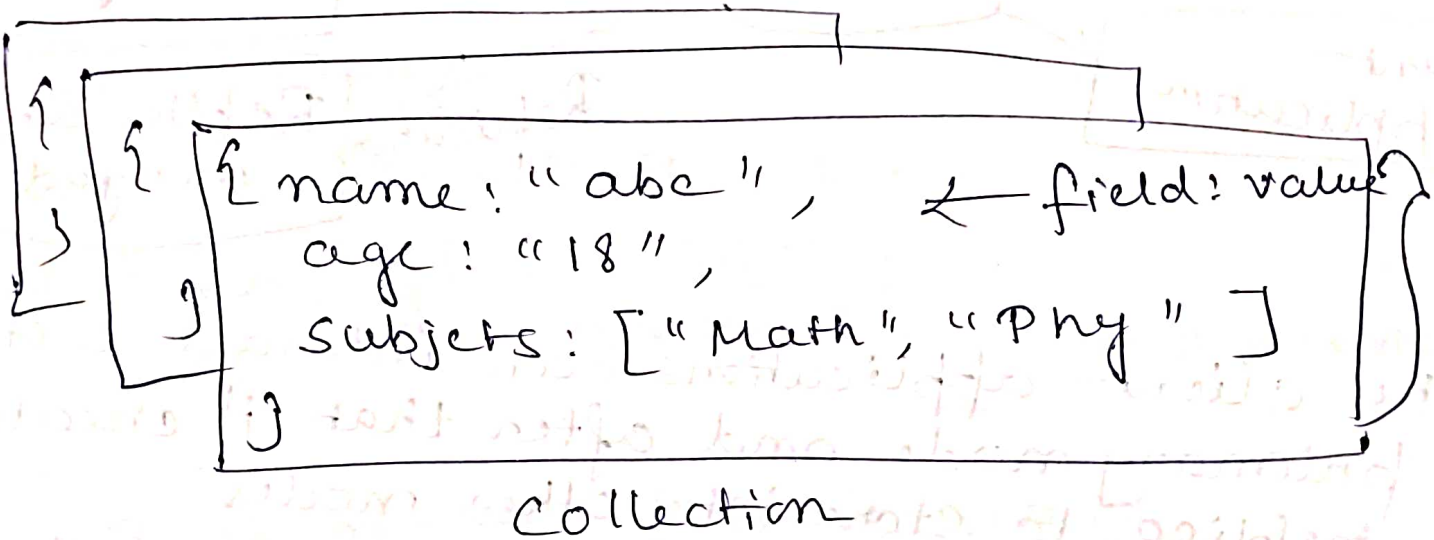
"mongo" is the command-line shell that connects to a specific instance of mongoD

"mongoD" is the "MongoDaemon",
It is basically the host process for database

mongo shell = Java script interface of mongoDB
- used to query and update data
- Performs CRUD operation

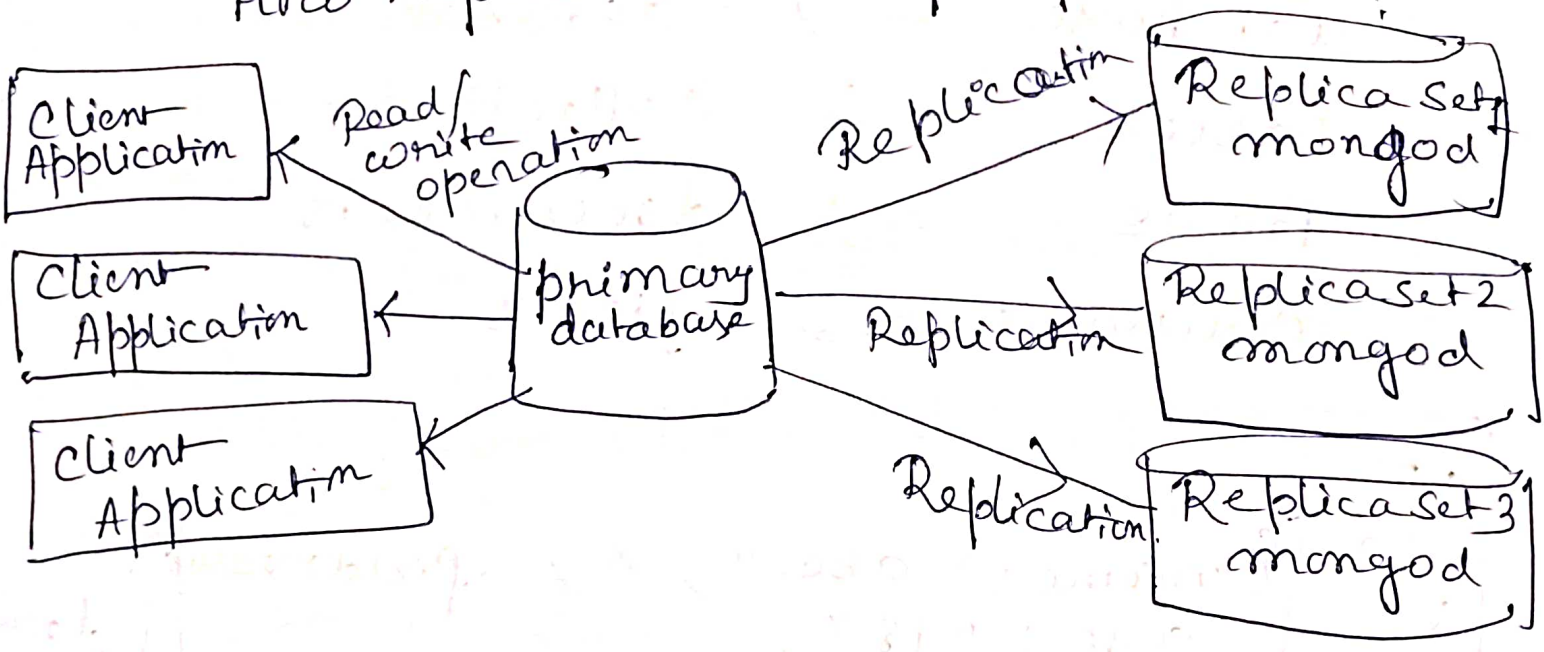
mongo compass
- GUI of MongoDB
- Interacts with data with CRUD operation.

SQL Terms		MongoDB Terms
database	⟹	database
tables	⟹	collections
rows	⟹	documents
columns	⟹	fields



Replication in Mongo DB

How replication is performed?



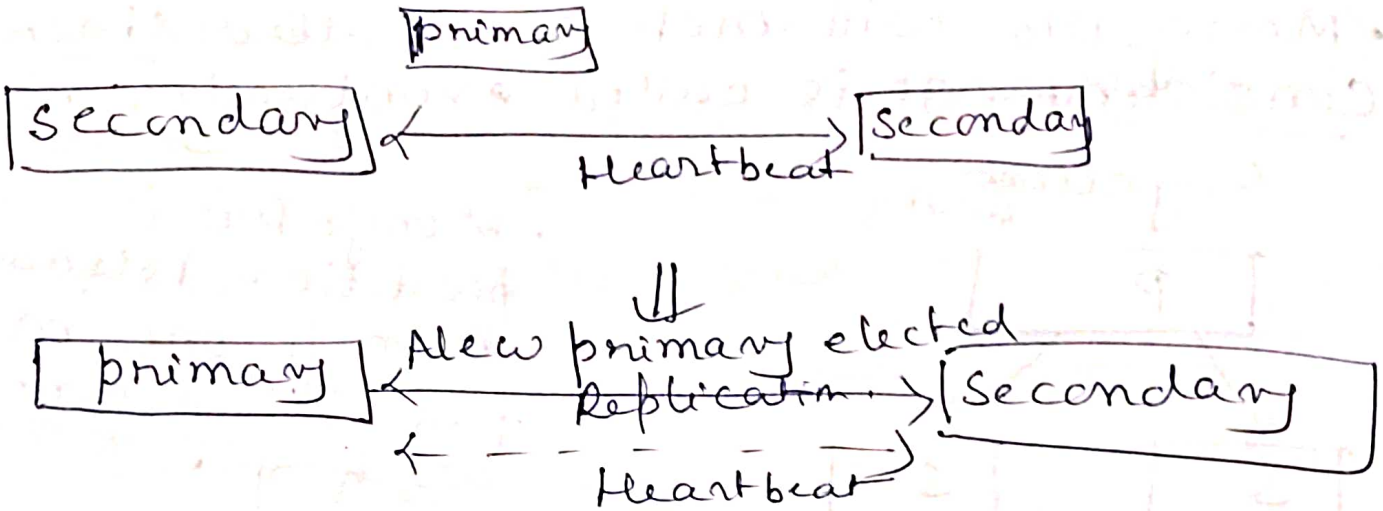
The client applications can interact with the primary node and after that it creates the replica to store into other nodes.

All the secondary nodes are connected with the primary node. There are one heartbeat signal from the primary node.

When the primary node goes down the, secondary nodes cannot get heartbeat signal.

The secondary nodes wait for 10 seconds for the signal, after that it can understand that the primary node is not working correctly.

After that it elects the new node as primary node.



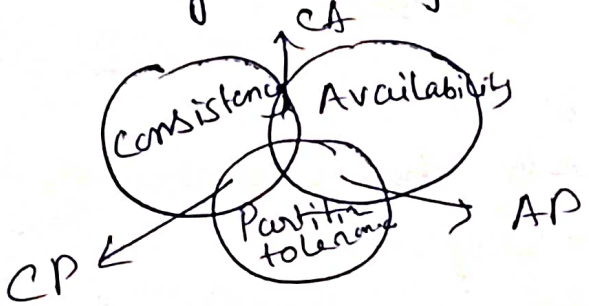
Advantage

(1) protect the data
(2) high availability
(3) No downtime for maintenance

CAP theorem of MongoDB

A distributed system has 3 properties

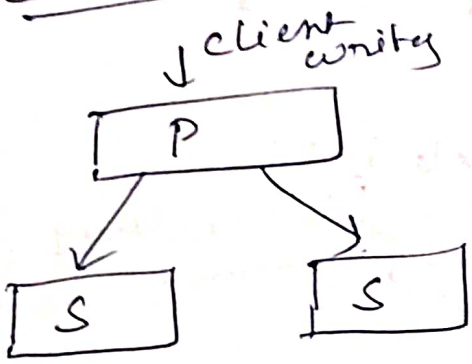
- Consistency :- All the nodes see the same data at same time.
- Availability :- Every request receives a response.
- Partition tolerance :- The system continues to operate even if part of the system fails.



MongoDB

(1) Consistency
(2) Partition tolerance

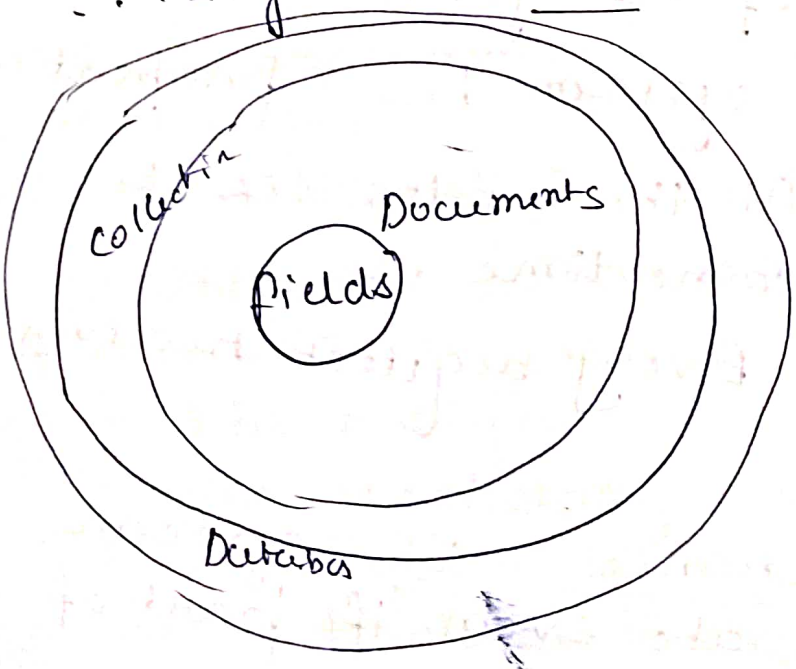
- MongoDB will not be called strongly consistent. It is called eventually consistent



- MongoDB is partition tolerant even if one node fails, it will continue working.

- MongoDB is mostly available, but the only time when the leader is down, MongoDB can't accept writes, until it figures out new leader. Hence, not highly available.

∴ MongoDB is CP



Mongo Shell

- Javascript interface of MongoDB
- It is used to query and update data
- Performs CRUD Operation.

Mongo compass

- GUI of MongoDB
- Interacts with data with CRUD operations.

Features :-

(i) MongoDB. Supports BSON

- BSON is binary variant of JSON
- BSON is faster to read, traverse and it contains more information.

(ii) Automatic shrading

Allows data to be spread across many systems.

(iii) Using NoSQL database design

BSON ⇒ Binary Object Notation.

- Not human readable format as it is in binary format.
- BSON is used to encrypt the JSON data.
- It provides additional data types.

Basic commands

(i) Use <db_name> ⟹ to create a database

(ii) show dbs ⇒ to show list of database

(iii) db ⇒ to show in which database you are now.

(iv) db.dropDatabase ⇒ to drop database.

MongoDB document

It consists of single / multiple set of key-value pairs.

```
{ name : "XYZ"
  category : "student"
  department : "CS"
  subjects : ["NoSQL", "SQL"]
}
```

MongoDB collection

A group of documents is referred to as "collection".

```
{ name : "abc",
  department : "CSE"
}
```

```
{ name : "XYZ"
  address : "patiala"
}
```

```
{ department : "CSE"
  address : "XXX"
```

Create collection

db.create collection ("employee")

db.{collection_name}.insert ("name": "xyz")

Capped collection

Fixed size collection.

While creating a collection, user must specify the collection's maximum size in bytes and the maximum number of documents that it would store

> db.create Collection ("users", { capped: true, size: 614200, max: 10000 })

to check whether it is capped collection or not

> db.~~capped~~ users.isCapped()

Drop collection

db.collection_name.drop()

CRUD operation

Four functions

(i) Create :- Adding new documents to the collection

(ii) Read :- Retrieving documents from collection.

(iii) Update :- Update documents in collection.

(iv) Delete :- Removing documents from collection.

Create

(1) Add/insert new document in collection

(2) Uses insert() method.

(3) If the collection does not exist, where a document is required to be added, insert() will create new collection.

```
> db.users.insert({"name" : "xyz"})
> db.users.insert({"name" : "xyz",
                   "email" : "sm@gmail.com"})
```

```
> db.users.insert([{"name":"sumana"},
                   {"name":"Maiti",
                    "email":"abc@gmai
                                    .e
```

_id :- act as primary key

Object id

```
> db.users.insert({"name":"xy2",
                   _id:"E001"})
```

```
> db.users.insertOne({field:"value"})
```

insertMany()

- Insert multiple documents in collecti
- Array of multiple documents is pas
- Both insertOne(), insertMany() wo return newly inserted documents wi _id field.

```
> db.course.insertMany([{item:"Apple",
                         qty:15},
  {item:"Orange", qty:10},
  {item:"mango", qty:20}
])
```

Create

- Add/insert new document in collection
- Uses insert() method
- If the collection does not exist where a document is required to be added, insert() will create new collection.

```
> db.users.insert({"name": "mark"})
> db.users.insert({"name": "paul",
                   "email": "sm@gmail.com"})
```

~~> db.~~

Insert() can also be used to insert multiple documents we can pass array as a parameter in find() method

```
> db.users.insert([{"name": "sumana"},
                   {"name": "Maite"}])
```

Insert can be used to insert single or multiple documents. Each document stored in a collection required a unique-id field that act as a primary key

Use of insertOne()

- insertOne() adds a single document in the collection

```
> db.<collection_name>.insertOne
                    ({"field": "value"})

> db.myCollectin.insertOne({name:
                              "XYZ
```

If the _id field is not specifie MongoDB assigns the object id by itself automatically for missing _id field

Use of insertMany()

- Insert / adds multiple documents in the collectin
- Array of multiple documents is passed through insertMany() method.
- Both insertOne() and insertMany() would return newly inserted document having newly created _id field values

> db.collection.insertMany([array]);

> db.course.insertMany([{item: "card", qty: 15}, {item: "envelope", qty: 20}, {item: "stamps", qty: 30}]);

(#) insert (method from older versions)

takes a single document by default, and there is an option to insert multiple documents supplied as an array

Read / Query Operations :-

Read through documents and retrieve the specified documents from a collection by querying the collection for documents. MongoDB provides find() method to read and retrieve documents from a collection.

```
> db.collection.find({field: "value"})
> db.course.find({module: "Java"})
```

Single or multiple filters.

Query creating filters using operation

A query to retrieve a specific document can be created using many operators provided by MongoDB.

eg $in

One can retrieve all documents in the particular collection where the field equals to specified value or either of multiple specified value.

db.<collection_name>.find({ field: { $in: ["value"]}})

↳ equivalent to

db.collection.find({ field: "value" })

Query documents having values equal to either of specified multiple value

db.<collection name>.find({field: {$in: ["value1", "value2"]}})

eg. Find the documents having module either Javascript or MongoDB

```
db.course.find({module:{$in:["Javascript","MongoDB"]}})
```

Query using logical operator

① $and :- Returns all documents that match the conditions of both clauses

```
db.users.find({"$and":
    [{"name":"mark"},
     {"age":30}]})
```

② $or :- Returns all documents that match the conditions of either clause

```
db.users.find({"$or":[{"name":"mark"},
    {"age":30}]})
```

(3) $not :- Inverts the effect of a query expression and returns documents that do not match query expression

```
db.product.find({price:{$not
                :{$gt:95
                }}})
```

Comparisons Operation

$eq :- Returns true if matches the documents where the value of the field equals to specified value

```
{<field> : { $eq : <value> }}
```

eg `db.product.find({qty:{$eq:20}})`

↑ Select all documents where the value of qty field equals 20

`db.product.find({qty:20})`

$ne

$ne selects documents where the value of the field is not equal to the specified value.

```
{field : {$ne : value}}
```

e.g. db.product.find({quantity :{$ne :20}})

← Select all documents in the product collect where quantity is not equal to 20

* This query will also select documents that do not have quantity field

$gt

{field : {$gt : value}}

Selects those documents where the value of the field is greater than (>) the specified value.

db.products.find({ quantity: {$gt: 20}})

$lt

{field : {$lt : value}}

Selects the documents where the value of the field is less than (<) the specified value

db.product.find({ quantity: {$lt: 20}})

$gte:

{field : {$gte : value}}

$gte selects the documents where the value of the field is greater than or equal to (>=) a specified value.

db.product.find({quantity: {$gte:20}})

$lte

{field : {$lte : value}}

$lte ~~helps~~ selects the documents where the value of the field is lesser than or equal to (<=) a specified value.

db.course.find({tags: "frontend development", time_days: {$lte:15}})

↑ Retrieve the document for "frontend development" module being completed in less than or equal to 15 days

Element operators

~~Matt~~

$exist

Matches documents that have the specified field.

eg Consider you have users collection with users information. Find all the users where phone no field exists

```
> db.users.insert({"num":"sum",
                   "phone":99})

> db.users.find({"phone":{"$exists
                          :true}
```

$type

Select documents if a field is of the specified type

Eg Consider you have user collection with user information.

Find all the users where phone field type is string

```
> db.users.insert({"name": "smith
                   "phone": 9999
```

```
> db.users.find({"phone":{"$type":
                  "string
                }})
```

Array Query Operators

$elemMatch

The $elemMatch operator matches documents that contain an array field with at least one element that matches all the specified query criteria.

```
{ <field> : { $elemMatch : { <query1>,
                             <query2>,
                             ... } } }
```

eg

```
{ _id : 1, results : [82, 85, 88] }
{ _id : 2, results : [75, 88, 89] }
```

The following query matches only those documents where the results array contains at least one element that is both greater than or equal to 80 and less than 85:

```
db.scores.find( { results : { $elemMatch :
                              { $gte : 80,
                                $lt : 85 } } } )
```

output

{ "_id" : 1 , "results" : [82, 85, 88] }

82 is both greater than or equal to 80 and less than 85

$all

The $all operator selects the documents where the value of a field is an array that contains all the specified elements.

{ <field> : { $all : [<value 1> , <value 2> ...] } }

eg { tags : { $all : ["ssl" , "security"] } }

is equivalent to

{ $and : [{ tags : "ssl" } , { tags : "security" }] }

```
{ _id : 1 ,
  code : "xyz" ,
  tags : ["book", "pen", "bag", "headphone"]
}

{ _id : 2 ,
  code : "abc" ,
  tags : ["book", "pen", "school"]
}

{ id : 3 ,
  code : "efg" ,
  tags : ["book", "~~bag~~ pen", "school", "bag"]
}
```

```
db.product.find({ tags :
{$all : ["book", "pen", "bag"]}})
```

Output

```
{ _id : 1 ,
  code : "xyz",
  tags : [ "book", "pen", "bag",
           "headphn" ]
}

{ _id : 3 ,
  code : "efg",
  tags : [ "book", "pen", "school",
           "bag" ]
}
```

$size: The $size operator matches any array with the number of elements specified by the argument.

eg Find documents where tag field has 3 elements

```
> db.temp.find({ "tags" : { "$size" : 3 } })
```

Update Operations

- Modified created documents in the collection
- These operations update a single collection at a time.
- Criteria or filters can be specified identifying the documents that are to be updated.
- Setting up the filters makes use of the same syntax used while executing read operations

MongoDB offers the following methods for updating documents.

1. The updateOne()
2. The updateMany()
3. The replaceOne()

$ currentDate :

Sets the value of a field to current date, either as a Date or a Timestamp.

$inc : Increments the value of the field by the specified amount.

$min : Only updates the field if the specified value is less than the existing field value

$max : Only updates the field if the specified value is greater than existing field value.

$mul :- Multiplies the value of the field by the specified amount.

$rename :- Renames a field

$ set : Set the value of a field in a document

```
db.course.updateOne {
    { module : "MySQL" }
    { $set : { module : "RDBMS",
               time_days : 3 }
      $currentDate : { lastModified
                       : true }
    }
}
```

```
> db.course.find (
    "_id" : "003"
    "module" : RDBMS",
        "time_days" : 3,
        "tags" : "BD",
        "lastModified" :
                ISODate("...")
```

we want to update time, where
~~and~~ module RDBMS
time_days : 3

UpdateOne() updates the first matching documents in the collection that matches the filter.

Using UpdateMany()

Let's say if we want to add a description to all the documents having tags of "Frontend Development"

```
> db.course.updateMany(
    { "tags" : "Frontend Development" },
    { $set : { description : "Basic coding" },
      $currentDate : { lastModified : true }
    }
)
```

Using replaceOne()

- replaceOne() will replace the first whole document which contain the specified fied value
- We cannot change _id value using replaceOne() as _id is immutable.

. .

change the name of MongoDB module and add description to it

```
db.course.replaceOne(
    { module: "MongoDB" }
    { module: "MOSQL",
      time_days: 5,
      tags: "BD",
      description: "Basic database design" }
)
```

Delete Operations

- Removes the specified documents from the collection.
- These operations delete documents in a single collection at a time
- Criteria or filters can be specified identifying the documents to be removed.

deleteOne() deleteMany()

* For remove single document from a collection

db.course.deleteOne({<field> : <value>})

db.course.deleteOne({module: "Javascript"})

* db.course.deleteMany({tags: "BD"})

$currentDate

We are updating the value of joiningDate field of an employee's document whose first name is om

```
> db.Employee.updateOne
        ({"name": "om"},
            {$currentDate: {joiningDate
                : true}})
```

```
{ "_id": "001"
  "name": "om"
  "department": "Development"
  "joiningDate": ISODate("2023-02-  -  ")
}
```

We are adding a new date field whose value is assigned by $currentDate operator.

```
> db.Employee.updateOne({"name": "om
        {$currentDate: {joiningDate
            : {$type: "date
        }}})
```

$inc

The $inc operator increments a field by a specified value.

eg

```
db.products.insertOne(
  { _id : 1,
    name : "abc",
    quantity : 10,
    metrics : { orders : 2, ratings : 3.5 }
  }
)
```

```
> db.products.updateOne(
  { name : "abc" },
  { $inc : { quantity : -2,
             "metrics.orders" : 1 } }
)
```

- increase the "metrics.orders" field by 1.
- increase quantity by −2 (decreases quantity)

```
quantity : 8
metrics : { orders : 3, ratings : 3.5 }
```

$min :- If updates the value of the field to a specified value if specified value is less than the current field value.

```
{ _id: 1
  highScore: 800
  lowScore: 200
}
```

$min compares 200 to specified value 150 and updates the value of lowScore to 150 as 150 < 200.

```
db.scores.updateOne({_id:1},
                    {$min:
                      {lowScore:
                        150}})
```

if

```
{ _id: 1
  highScore: 800
  lowScore: 150
}
```

```
db.scores.updateOne({_id:1},
                    {$min:{lowScore:250
                    }})
```

⇒ {_id:1, highScore:800, lowScore:150
}

$max :- The $max operator updates the value of the field to a specified value if the specified value greater than the current value of the field.

```
{ _id : 1
  highscore : 800
  lowscore : 200
}
> db.scores.updateOne ( { _id : 1 },
                        { $max : { highscore
                                   950
```

output

```
{ _id : 1
  highscore : 950
  lowscore : 200
}
> db.scores.updateOne ( { _id : 1 },
                        { $max : { highsc
                                   : 7
                                 } }
```

output

```
{ _id : 1
  highscore : 950
  lowscore : 200
```

$mul

Multiply the value of a field by a number.

```
{"_id": 1
 "item": "Hats",
 "price": 200$,
 quantity : 25
}
```

multiply price by 2 and quantity field by 3

```
> db.products.updateOne(
    {"_id": 1}
    { $mul:
        { price: 2
          quantity: 3
        }
    }
)
```

$rename:

The $rename operator updates the name of a field

```
> db.students.updateOne(
    { _id: 1 },
      { $rename : { 'cell' : 'mobi
                              } }
    )
```

It renames the field cell to mobi

$push

- The $push operator appends a specified value to an array
- If the field is absent in the document to update, $push adds the array field with the value as its element
- If the field is not an array, the operation will fail.
- If the value is an array, $push appends the whole array as a single element.

To add each element of the value separately, use the $each modifier with $push

```
{ _id : 1
  scores : [44, 78, 38, 80]
}
```

append a value to an array

```
> db.students.updateOne(
        {_id : 1},
        { $push : { scores : 89 } }
    )
```

output: { _id : 1, scores : [44, 78, 38, 80, 89] }

Append a value to arrays in multiple documents

```
{ _id : 2
  scores : [45, 78, 38, 80, 89]
}
{ _id : 3
  scores : [46, 78, 38, 80, 89]
}
{ _id : 4
  scores : [47, 78, 38, 80, 89]
}
```

The following $push operation appends 95 to the scores array in each document.

```
> db.students.updateMany(
        { },
        { $push : { scores : 95 } }
    )
```

Append Multiple values to an array

use $push with $each modifier to append multiple values to array field

```
>db.students.updateOne(
  { name: "joe" },
  { $push: { scores: { $each: [90,92,85
                              ]
```

Use $push Operator with multiple modifier

```
{ "_id" : 5,
  "quizzes" : [
      { "wk" : 1, "score" : 10 },
      { "wk" : 2, "score" : 8 },
      { "wk" : 3, "score" : 5 },
      { "wk" : 4, "score" : 6 }
  ] })
```

```
>db.students.updateOne(
    { _id: 5 },
    { $push: {
        quizzes: {
            $each: [ { wk: 5, score: 8 },
                     { wk: 6, score: 7 },
                     { wk: 7, score: 6 } ],
            $sort: { score: -1 },
            $slice: 3
        } } }
```

- the $each modifier add multiple documents to quizzes only
- the $sort modifier to sort all the elements of the modified quizzes array by the score field in desce order.
- the $slice modified keeps only first three sorted elements of qui array

output
```
{ "_id": 5,
  "quizzes": [
      { "wk": 1, "score": 10 },
      { "wk": 2, "score": 8 },
      { "wk": 5, "score": 8 }
```

$pop

$pop operator removes the first or last element of an array.

Pass $pop a value [-1] to remove the first element of the array

Pass [1] to remove the last element of the array.

- The $pop operation fails if the <field> is not an array.
- ~~The $pop~~

```
{ _id: 1
  scores: [8, 9, 10]
}
> db.students.updateOne({_id: 1},
                        { $pop:
                          { scores: -1
                          } })
```

output:

```
{ id: 1,
  scores: [9, 10]
}
```

```
{ _id : 10
  scores : [9, 10]
}
```

```
> db.students.updateOne ({_id : 10},
                         {$pop : {scores : 1}})
```

output :

```
{ _id : 10,
  scores : [9]
}
```

$each :- Multiple values will be appended to array field.

eg
>db.stu.updateOne({name: "joe"},
{$push: {scores: {$each: [90,92,95]}}})

It appends each element of [90, 92, 85] to the scores array for the document where the name field equals to joe

$slice :- The number of array elements is limited. It requires the use of $ each modifier.

~~>db.student~~.

{ $push: {
$each: [<value1>, <value2>, ...],
$slice: <num>
}
}

The <num> can be

value	description
zero ⇒	to update the array field to empty array
negative ⇒	to update the array field to contain only last <num> elements.
positive ⇒	to update the array field contain only first <num> elements.

eg

{"_id":1, "scores": [40, 50, 60]}

```
>db.students.updateOne(
    {_id:1},
    {$push:{
        scores:{
            $each:[80,78,86],
            $slice:-5
        }}}
```

It adds new elements to the scores array and then it uses the $slice modifier to trim the array to the last 5 elements

{"_id":1, "scores": [50, 60, 80, 78, 86]}

eg

```
{ "_id" : 2,
  "scores" : [89, 90]
}
```

```
> db.students.updateOne(
      { _id: 2 },
      { $push: {
          scores: {
              $each: [100, 20],
              $slice: 3
          }
      }
  })
```

```
{ "_id": 2, "scores": [89, 90, 100] }
```

update array using slice only

```
{ "_id" : 3, "scores" : [89, 70, 100, 20] }
```

```
> db.students.updateOne(
      { $push: {
          scores: {
              $each: [ ],
              $slice: -3
          }
      }
  })
```

output :

```
{ "_id" : 3,
  "scores" :
      [70, 100, 20]
}
```

$sort

Elements of the array are to be sorted. It requires the use of the $each modifier

specify [1] ⇒ ascending
[-1] ⇒ descending.

```
{"_id": 1,
 "quizzes": [ { "id": 1, "score": 6 },
              { "id": 2, "score": 9 }
            ]}
```

```
> db.students.updateOne(
    { _id: 1 },
    { $push: {
        quizzes: {
          $each: [ { "id": 3, score: 8 }
                   { "id": 4, score: 7 }
                   { "id": 5, score: 6 } ]
          $sort: { score: 1 }
        }}})
```

output

```
"quiz" [
        { id : 1 , score : 6 } ,
        { id : 5 , score : 6 } ,
        { id : 4 , score : 7 } ,
        { id : 3 , score : 8 } ,
        { id : 2 , score : 9 }
    ]
}
```

sort array elements that are not documents.

```
> db.students.insertOne( { "_id" : 2
                           "tests" : [89, 70,
                                      89,
                                      50] } )
```

```
> db.students.updateOne(
        { _id : 2 },
        { $push : { tests : { $each : [40, 60],
                              $sort : 1 } } } )
```

output

```
{ _id : 2 , tests : [ 40, 50, 60, 70, 89, 89 ] }
```

```
{ _id : 3
  tests : [89, 70, 100, 20]
}
```

```
>db.students.updateOne(
        { _id : 3 },
        { $push : { tests : { $each : [ ],
                              $sort : -1 } } }
)
```

output

```
{ "_id" : 3 ,
  "tests" : [100, 89, 70, 20]
}
```

$position

$position :- the location in the array at which to insert the new element is specified.

It requires the use of $each modifier.

Without the $position modifier, the $push appends the elements to the end of the array

```
{
  $push : {
    <field> : {
      $each : [<value 1>, <value 2>, ...]
      $position : <num>
    }
  }
}
```

<num> indicates the position in the array based on a zero-based index.

- If <num> is greater or equal to the length of the array, the $position modifier has no effect and $push adds elements to the end of array

- A -ve number corresponds to the position in the array.
  Counting from the last element of the array.

'-1' indicates the position just before the last element in the array.

```
{"_id" : 1
 "scores" : [100]
}
```

```
db.students.updateOne(
    { _id: 1},
    { $push: {
        scores: {
            $each: [50, 60, 70],
            $position: 0
        }}})
```

output

```
{ "_id" : 1 , "scores" : [50, 60, 70, 100
```

```
{ "id" : 2
  "scores" : [60, 60, 70, 100]
}
```

```
> db.students.updateOne(
    { _id : 2},
    { $push : {
        scores : {
          $each : [20, 30],
          $position : 2
        } } })
```

output

```
{ "_id" : 2, "scores" : [60, 60, 20, 30, 70, 100] }
```

```
> db.students.updateOne(
    { _id : 3},
    { $push : {
        scores : {
          $each : [90, 80],
          $position : -2
        } } })
```

output

```
{ "_id" : 3, "scores" : [60, 60, 20, 30, 90, 80, 70, 100] }
```

$pop

The $pop operator removes the first or last element of an array

[-1] ⇒ removes the first element of the array.

[1] ⇒ removes the last element of the array

~~> db.~~

```
{ _id: 1,
  scores: [8, 9, 10]
}
```

```
> db.students.updateOne({_id:1},
                        {$pop: {scores:
```

output:
```
{ _id: 1,
  scores: [9, 10]
}
```

```
> db.students.updateOne({_id: 1},
                        {$pop: {scores
                                :1}
```

output:
```
{ _id: 1
  scores: [9]
```

$pull

$pull operator removes from an existing array all instances of a value or values that match a specified condition.

```
{ _id : 1
  fruits : [ "apples", "oranges", "bananas"
  vegetables : [ "carrots", "tomato"
                                "carrots
}
{ _id : 1
  fruits : [ "kiwis", "oranges", "bananas
                                "apples" ],
  vegetables : [ "broccoli", "carrots",
                                "onions" ]
}
```

- Remove "apples" and "oranges" from fruits array.
- Remove "carrots from vegetables array.

```
> db.stores.updateMany(
    { },
    { $pull: { fruits: { $in: ["apples",
                               "oranges"] },
               vegetables: "carrots"
             } })
```

output

```
{ _id: 1
  fruits: ["bananas"]
  vegetables: [ tomato ]
}

{ _id: 2
  fruits: ["kiwi", "banana"]
  vegetables: ['broccoli', 'onions
}
```

```
{ _id : 1
  votes : [3, 5, 6, 7, 7, 8]
}
```

Remove all items from votes array that are greater than or equal to ($gte 6 :

```
> db.profile.updateOne( { _id : 1 },
                        { $pull : { votes :
                                    { $gte : 6
                                    } } } )
```

output

```
{ _id : 1
  votes : [3, 5]
}
```

$pullAll

- $pullAll operator removes all instances of the specified values from an existing array

$pull operator removes elements by specifying a query.

```
{ _id: 1
  scores: [0, 2, 5, 5, 1, 0]
}
```

```
> db.survey.updateOne({_id:1},
                      {$pullAll:
                        [scores: [0,5]
                        ]})
```

output

```
{ _id: 1
  scores : [2, 1]
}
```

Query an Array of Embedded Documents

```
{ item: "journal"
  instock: [ { warehouse: "A",
               qty: 5 },
             {
```

```
{ "_id": ObjectId("                    ")
  "firstName": "John",
  "LastName": "king",
  "email": "john@abc.com"
  "address": { "street":
               "house":
               "city":
               "country": "USA"
             }
}
```

```
> db.employee.find( { "address.city": "Patiala"
                    } )
```

Array of Embedded documents

```
{ item: "journal"
  stock: [ { warehouse: "A", qty: 5 },
           { warehouse: "C", qty: 15 } ]
},
{ item: "notebook",
  stock: [ { warehouse: "E", qty: 5 } ]
},
{ item: "paper",
  stock: [ { warehouse: "B", qty: 6 },
           { warehouse: "D", qty: 10 } ]
}
```

```
> db.inventory.find( {
        "stock": { warehouse: "B",
                   qty: 5 } } )
```

Nested documents

```
> db.inventory.insertMany([
{ item: "journal", qty: 25, size: { h: 14, w: 21 }, status: "A" }
{ item: "notebook", qty: 50, size: { h: 8, w: 11 }, status: "A" }
{ item: "paper", qty: 100, size: { h: 8.5, w: 11 }, status: "D" }
{ item: "postcard", qty: 45, size: { h: 10, w: 15.25 }, status: "A" }
]);

> db.inventory.find( { size: { w: 21, h: 14 } } )

> db.inventory.find( { "size.h": 8 } )
> db.inventory.find( { "size.h": { $lt: 15 } } )
```

Query an Array of embedded documents

```
>db.inventory.insertMany([
 { item: "journal", instock: [ { warehouse: "A",
                                 qty: 5 },
                               { warehouse: "C",
                                 qty: 15 } ] },
 { item: "notebook", instock: [ { warehouse:
                                  "C",
                                  qty: 5
                                } ] },
])
```

```
> db.inventory.find( { "instock":
                       { warehouse:
                         "A"
                         qty 5 } } )
```

Selects all documents where an element in the instock array matches the specified document

```
> db.inventory.find( { "instock": { qty: 5
                                    warehouse
                                    : "A
```

Requires exact match (including order)

> db.inventory.find({'instock.qty': {$lte:20}})

// at least one embedded document that contains the field qty whose value <=20

> db.inventory.find({"instock": {$elemMatch: {qty:5, warehouse:"A"}}})

```
{ item: "journal"
  status: "A"
  size: { h: 14
          w: 21
        }
  instock: [ { warehouse: "A",
               qty: 5 } ]
}
```

```
> db.inventory.find ( { status: "A" } )
```

Select * from inventory where status = "A"

Return specified fields and _id field only

```
> db.inventory.find ( { status: "A" },
                      { item: 1, status: 1 } )
```

Select _id, item, status from inventory where status = "A"

You can remove the _id field from the results by setting it to 0

```
> db.inventory.find ( { status: "A" }, { item: 1,
                                         status: 1,
                                         _id: 0 } )
```

Select item, status from inventory where status = "A"

Return all but the excluded fields

```
> db.inventory.find({ status : "A" },
                    { status : 0,
                      instock : 0 })
```

Return all fields except for status and instock fields.

```
> db.inventory.find(
    { status : "A" },
    { item : 1, status : 1,
      "size.h" : 1 }
)
```

Suppress specific fields in embedded documents

```
> db.inventory.find({ status : "A" },
                    { "size.h" : 14 })
```

Projection on embedded documents in an array

```
> db.inventory.find ({status : "A"},
                     { item : 1, status : 1,
                       "instock.qty" : 1})
```

Projection operator

- $elemMatch
- $slice
- $

```
> db.inventory.find ({ status : "A"},
                     { item : 1,
                       status : 1,
                       instock : {$sl
```

↳ Return the last element in instock array.

Return all but the excluded fields

```
> db.inventory.find({ status : "A" },
                    { status : 0,
                      instock : 0 })
```

Return all fields except for status and instock fields.

```
> db.inventory.find(
        { status : "A" },
        { item : 1, status : 1,
          "size.h" : 1 }
)
```

Suppress specific fields in embedded documents

```
> db.inventory.find({ status : "A" },
                    { "size.h" : 14 })
```

Projection on embedded documents in an array

```
> db.inventory.find ([{status : "A"},
                      { item : 1, status : 1,
                        "instock.qty" : 1
```

Projection operator

- $elemMatch
- $slice
- $

- > db.inventory.find ([{ status : "A"},
                        { item :"1,
                          status : 1,
                          instock : {$

↳ Return the last element in instock array.

```
{ item : "Diary",
  stock : [ { storehouse : "X",
              qty : "15" },
            { warehouse : "2",
              qty : 25 } ],
  _id : "001"
}

{ item : "Register",
  stock : [ { storehouse : "X",
              qty : 10 } ],
  status : "B",
  _id : "002"
}

{ item : "Chartbook",
  stock : [ { storehouse : "X",
              qty : 40 },
            { storehouse : "y",
              qty : 25 } ],
  _id : "003"
}
```

> db.course.find({"stock":
{storehouse:"y",
qty:25}})

By default it will retrieve all the fields.

> Select * from course

> Select item from course

condition

> db.{collection_name}.find({field1:"valx"},
{field1:1,
field2:1
})

only field1 and field2 will be retrieved.

But by default -id field will be automa-tically retrieved.

· The field of -id needs to be specifically set as "0"

> db.course.find({"stock.storehouse":"Y"},
{"stock.storehouse":1})

> db.course.find({"stock.storehouse":"Y"},
{stock:1, -id:0})

limit()

If we do not want all the documents, we are interested in first2 then limit() method will be used.

```
> db.course.find({"stock.storehouse":"Y"},
                 {stock:1, _id:0})
  .limit(2)
```

$elemMatch

It returns only the first element matching the specified condition.

Index

- Indexes are used to support efficient execution of queries.
- Without indexes, MongoDB perform a collection scan, i.e scan every document in a collection, to select those documents that match query statement.
- 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 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 value of the field.

Default _id index

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

The `_id` index prevents clients from inserting two documents with same value for _id field. We can not drop the index on `_id` field

Create an idex

> db.<collection_name>.createIndex({name:-1})

It will create an index on name field in descending order.

> db.<collection.name>.createIndex({rollno:1})

1: Ascending order
2. Descending order.

Types of indexing

(1) Single field index
(2) Compound index
(3) Multikey index
(4) Geospatial "
(5) Text "
(6) Hash "

Single field

Record
{ "_id" : objectId (" ")
"score" : 1024
"location" : { "state : Punjab",
"city" : Patiala" }
}

To create ascending order

> db.record.createIndex ({"score" : 1})

Single field index is used to generate an index on a document's single field.

Create index on embedded field

> db.records.createIndex({ "locati
state
: 1

Compound index :-

It allows you to create user defined index on several fields. MongoDB uses compound index for this.

If a compound index has the elements. "name" : 1, "course" the index will sort the name first then the course

```
{ "_id":
  "item" : "orange"
  "category" : ["food", "produce",
                "grocer
  "location" : "Tiet",
  "stock" : 4
}
```

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

The order of the fields listed in a compound index is important

```
> db.events.find().sort({username:1,date:-1})
```

first ascending username values and then by descending of date values.

Multikey index

To index a field that holds array value, MongoDB creates an index key for each element.

```
> db.collection.createIndex({field_name:1/-1})
```

If an indexed field is an array then MongoDB automatically create a multikey index for that field.

You do not need to explicitly specify the multikey type.

You cannot create a compound multikey index if more than one index field of a document is an array.

```
{ _id : 1,
  a : [1, 2],
  b : [1, 2],
}
```

we cannot create compound multikey index {a:1, b:1} as both a and b are array field.

```
{ _id : 1, a : [1, 2], b : 1 }
{ _id : 2, a : 1, b : [1, 2] }
```

A compound multikey index {a:1, b:1} is permissible as for each document, only one field indexed by compound multikey index No document contains array values for both a and b fields.

Geospatial index

It helps to find location near another location.

```
{ name : "central Park"
  location : { type : "Point",
               coordinates : [-73.99, 40.77] }
  category : "Park"
}

{ name : "SD Park",
  location : { type : "Point",
               coordinates :
                   [-73.99, 40.71] }
  category : Park
}
```

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

```
> db.collection.find(
    { location
      { $near:
        { $geometry
          : { type: "point",
              coordinate:
                [-73.99,
                 40.78
          $minDistance: 1000
          $maxDistance: 150
    }}})
```

GeoJson

The following example specifies GeoJSON point

```
location: { type : "Point"
            coordinates : [40, 5]
          }
```

GeoJSON LineString

```
location: { type : "LineString"
            coordinates : [[40, 5], [41, 6]] }
```

To specify GeoJSON data, use an embedded document with

- A field named 'type' that specifies GeoJSON object type.
- A field names 'coordinates' that specifies the objects coordinates.

If specifying latitude and longitude coordinates, list longitude first, then latitude.

- Valid longitude values are between -180 and 180, both inclusive.
- Valid latitude values are between -90 and 90, both inclusive.

2dsphere indexed

A 2dsphere index supports queries that calculate geometries on an earth-like sphere.

The 2dsphere index supports data stored as GeoJSON objects

```
>db.collection.createIndex({<location field> : "2dsphe
```

The <location field> is a field whose value is either a GeoJSON object

```
> db.places.insertMany([
    { loc: { type: "Point",
             coordinates: [-73.97,
                           40.77] },
      name: "Central Park",
      category: "Parks"
    },
    { loc: { type: "Point",
             coordinates: [-73.88,
                           40.77] },
      name: "Delhi Airport",
      category: "Airport"
    }
])

> db.places.createIndex({ loc: "2dsphere" })
```

Create a compound
2dsphere Index key

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

> db.places.createIndex({ loc: "2dsphere", category: -1, name })

A compound 2d sphere index does not require the location field to be the first field indexed.

> db.places.createIndex({ category: 1, loc: "2dsphere" })

db.collection.find(
{ location: {
$near: {
$geometry: {
type: "point",
coordinates: [-73.85, 40.84]
},
$maxDistance: 1000
$minDistance: 10
}
}
})

returns documents from nearest to farthest

1000 = 1 KM

10 = 10 meters

$ geometry operator specifies a GeoJSON geometry for use with geospatial query operators:

- $ geoWithin
- $ geoIntersects
- $ near

Latitudes are horizontal
Longitudes are vertical line

Text index

{ id: "01"
  course: C
  descriptim: "

}

The data is stored in the foromat of the document, it can hold a huge amount of data. So searching is an importent criteria here and for that MongoDB provides Text indexes to support text search queries especially on string content.

~~Text index~~

> db.collectimName.createIndex ({field: "text"})

> db.collectimName.createIndex ({descriptim: "text"})

How to search using text index

> db.collectimName.find({ $text: { $search: "mongo" } })

You can index multiple fields for text index.

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

A compound index can include text index keys in combination with ascending / descending index key.

Hashed index

It maintains entries with hashes of the value as index field

```
> db.collection.createIndex
    ({ _id : "hashed" })
```

MongoDB automatically creates the hashes

Database Sharding

- fundamental concept of very large scale system
- Scale out database as much as possible

Different options of scaling database

Sharding is optimization technique to achieve horizontal scaling in database.

Sharding is specific type of partitioning

options

(1) Scaling up hardware

(i) double your RAM
if 16 then make 32

(ii) use better processor.

It is very expensive to scaling up and up and up.

If you scale the ~~performance~~ power of machine, it is not necessary that you double the performance ~~to~~ boost.

There is a technical maximum that how much you can scale up a given machine.

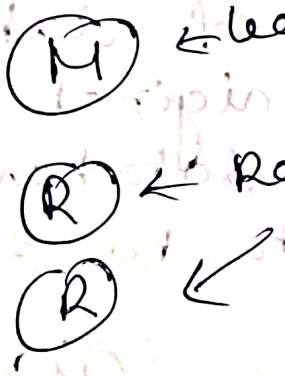
∴ So definitely some constraint

2, 8. Add replicas (Read replicas)

M

You have copies of database. Instead of having one copy, you have multiple copies of database. copies can also handle multiple traffic.

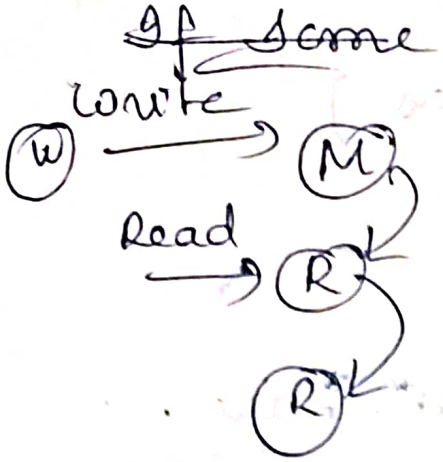
So nosql instead of your one database getting overburdened with all read and all write requests, it splits into multiple replicas.



Master node is responsible to receive all write requests.

The read replicas are not allowed to receive any write requests.

Master will response to write request and propagate the update to that down to read replicas. So update eventually The problem is eventual consistency



← Master node commits the request

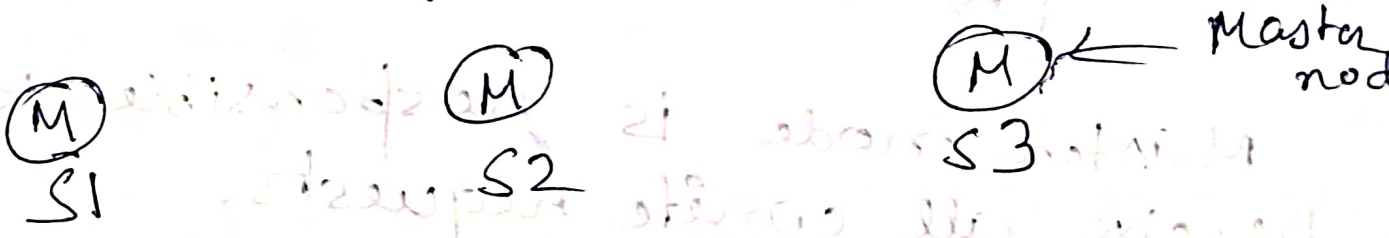
Eventually consistent

This will create problem.

(iii) Sharding

Split your database into multiple smaller databases.

Let we have 3 databases and all contains subset of database. We separate the original database into these smaller databases and allocate different data to each one.



How we can divide the data; fundamental idea is using a key. Key is input to hashing function and this key will tell you which shard the data regarding this key reside.

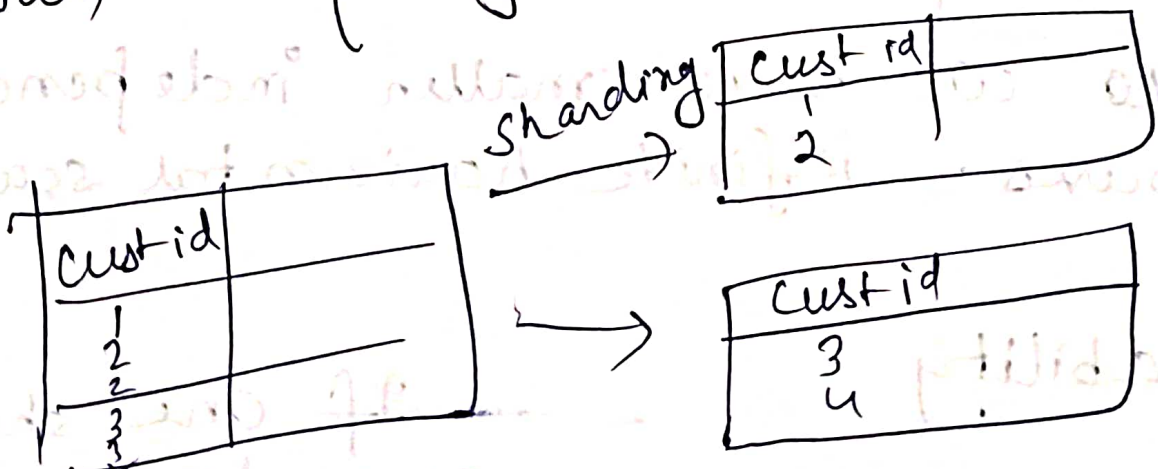
customer id	invoice id	date
1		
2		
3		
3		
4		

Horizontal partitioning

When the query comes, how do you know in which database the information reside.

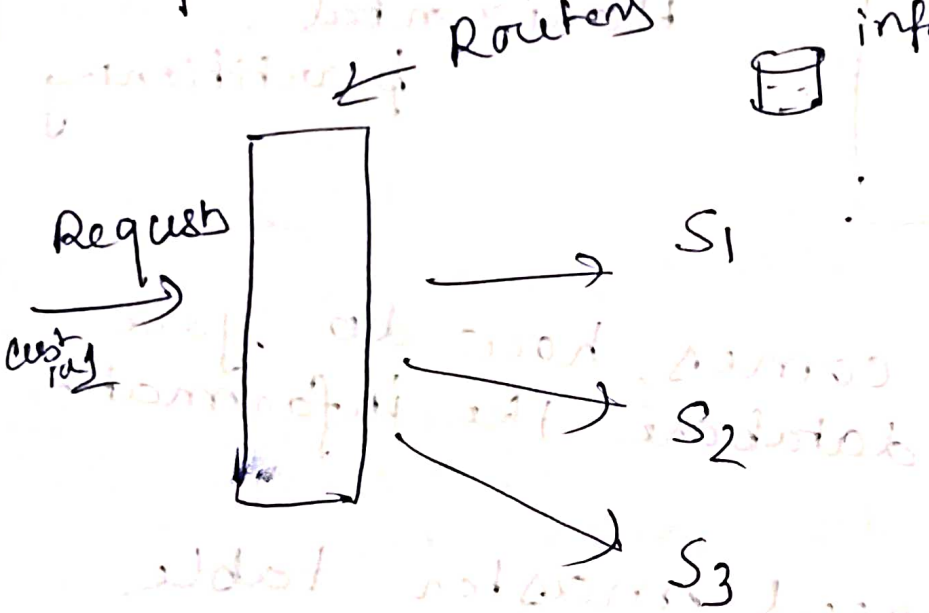
Either you need master table which tells this customer in this shard or you need some hashing function you can use like some algorithmic approach

If only single table, then if single point of failure, whole system goes down, no query will work.



Now your data is splitted up into different databases.

We add some intermediary layer that says which shard contains which information.



Now the problem is you have additional layer of complexity

Pros

1) Scalability
   - we have smaller independent databases. infinite horizontal scaling

2) Availability & Fault tolerance
   - If one shard goes down, other application can still goes

Cons

(1) It is expensive

(2) Complexity → Partition mapping
(1) Algorithmic approach
{ cust id → shard no }
(2) Database
cust id in shard
etc etc

Introduce routing layer
{ This does not come free }

Non uniformity
{ 100 → 50 50
uneven storage
Resharding data
(More complex

2)

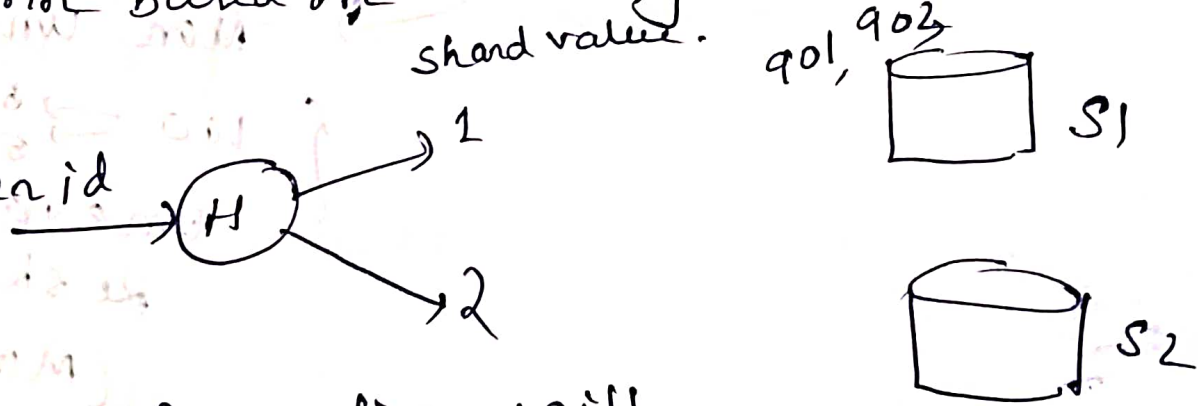
Key based sharding

User id	Name	City
901	A	Berlin
902	B	Dubai
903	A	Delhi
904	C	Mumbai

First is we have to choose a key. (Shard Key)

Shard key may not be primary key of a table

Choosing a shard key means you pick up a column based on which you divide data shard value.



Hash function will always give same result

Your application will tell in which shard your request will go

Algorithmic sharding.

We are calculating on the go

advantage

(1) Your data is evenly distributed

(2) Every shard you have will contain same amount of data

disadvantage

(1) If your data increases, you need to add new shard, then there is a problem as hashing function needs to be changed.

So now you may need to move your data from shard 1 to shard 2, shard 2 to shard 4.

(2) If you decrease the data, reduce some shard, there also some problem as hash function needs to be changed.

How to choose shard key

You should not choose a key which frequently changes.

If 'city' is shard key, then prob

It should be static

You can choose combination of columns as shard key

Range based sharding

U-id	event	date	
100	click-login	17-02-2021	ts1
101	click-buy	18-03-2020	ts2
102	click add to card	19-04-2020	ts3
103	click promo	20.04.2020	ts4

Jan Feb Mar Apr May

You have to store the data for anylytical purpose. The data needs to be available in terms of date or month

You can divide the data in terms of date/month. (on monthly basis)

I have 6 shards.

I can store the data montly basis Two month's data in one shard

Query :- Give all the login details of February

Only one shard we have to check and all the login details can be filtered out

We can store the data on price basis

100 – 200 — S1

201 – 300 — S2

This is called range based sharding.

Here we can store the data.

Here also we use application code

Advantage

As there are no hash function, we can use more machines.

Disadvantage

Let we have different shard but Jan Feb entry is more. This is called Hotspot.

∴ This may not be uniformly distributed

Data

user names

a to e : f to J

Queries like >, <, range base query it is useful

Directory based sharding

u id	country/zones
1	A
2	B
3	C
4	D

Lookup table

zones	shard
A	1
B	2
C	3
D	4

Dynamically add or delete entries

Extra layer to store data.

Single point of failure

Backup of lookup table