



DMIF, University of Udine

---

# Introduction to MongoDB 4

Paolo Gallo  
paolo.gallo@uniud.it

May 18, 2020



- 1 Overview of MongoDB
- 2 MongoDB Data Structures
- 3 MongoDB Indexes
- 4 Replication
- 5 Sharding
- 6 Aggregation
- 7 Bibliography

# Overview of MongoDB



MongoDB is a radical departure from relational databases as defined by Dr. Edgar F. Codd in his paper *A Relational Model of Data for Large Shared Data Banks*, published in 1970.

The first version of MongoDB was introduced in 2009 by a company named 10gen (now MongoDB Inc.)

Main goal was to address some needs about handling big data and modelling objects that were not yet available in RDBMS.



MongoDB is probably one of the world's most popular NoSQL database, in particular it is a *document oriented database*

Most companies use MongoDB in different application scenario, from IoT to Web:



**BOSCH**



ebay



Driver & Vehicle  
Licensing  
Agency



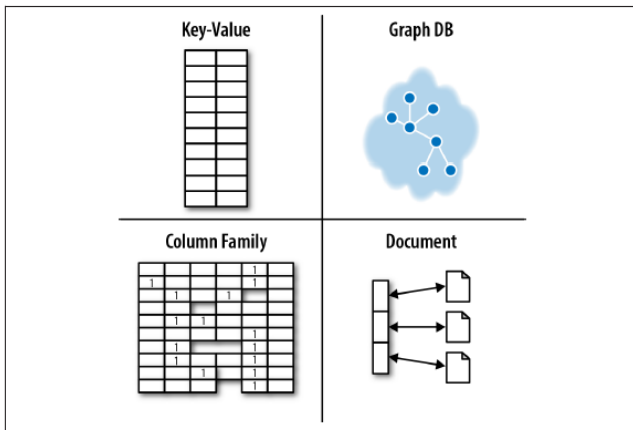
RDBMS systems are designed to minimise storage, an expensive resource 50 years ago.

RDBMS provide flexibility by creating relations between tables, but *normalisation* may introduce overheads when handling big data.

MongoDB addresses the needs of big data:

- avoiding a fixed data model
- parallel distributed processing algorithms such as MapReduce
- sharding allows fragments of a database to be stored on different servers

# The NoSQL Families Quadrant





# What is NoSQL?

NoSQL databases use other modelling paradigms:

- *Key/Value* Data is stored as keys and values, like a multidimensional array, values can be obtained by simply referencing the key
- *Graph* A database where basic entities are represented as nodes connected by edges, additional properties can be associated to both above entities
- *Column Oriented* exploits the benefits from the physical storage in columns rather than in records
- *Document oriented* Data is organised in documents, eventually documents can contain other documents in a XML fashion





# Document Oriented Stores

Document databases store and retrieve documents, just like an electronic filing cabinet

Documents tend to comprise maps and lists, allowing for natural hierarchies, similar to formats like JSON and XML.

At the simplest level, documents can be stored and retrieved by ID. Using a meaningful ID, they're like a key/value store



# The newer the better ?

## Warning

Please note that RDBMS are still widely used today and they're the most preferable choice for many applications.

Even if newer technologies, conceptually far from RDBMS, are available, a preliminary design process must find out out which technology suits better for a specific application.

Features supported by any RDBMS are simply *not supported* in MongoDB and vice versa

# MongoDB Data Structures



JavaScript Object Notation (JSON) is an open, human and machine-readable standard that, along with eXtensible Markup Notation (XML), facilitates data interchange

MongoDB represents JSON documents in binary-encoded format called BSON behind the scenes.

BSON extends the JSON model to provide additional data types, ordered fields, and to be efficient for encoding and decoding within different languages.



# XML Is Like JSON

XML is Like JSON Because:

- both are "self describing" (human readable)
- both are hierarchical (values within values)
- both can be parsed and used by lots of programming languages
- both can be fetched with an XMLHttpRequest



JSON is Unlike XML Because:

- JSON doesn't use end tag
- JSON is shorter (compact in size)
- JSON is quicker to read and write
- JSON can use arrays
- XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.



# XML & JSON Example

## XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

## JSON

```
{ "empinfo" :
  {
    "employees" : [
      {
        "name" : "James Kirk",
        "age" : 40,
      },
      {
        "name" : "Jean-Luc Picard",
        "age" : 45,
      },
      {
        "name" : "Wesley Crusher",
        "age" : 27,
      }
    ]
  }
}
```



# BSON vs. JSON

- A BSON data structure is an ordered object, JSON objects are not
- BSON does not require field names to be unique, JSON fields are unique
- BSON supports additional data types such as binary data (32-bit and 64-bit integers, floats, and Decimals), in JSON type Number is a double floating point
- MongoDB has defined comparison and sort order rules for BSON values





# BSON Data Types I

- String UTF-8 (string)
- Integer 32-bit (int32)
- Integer 64-bit (int64)
- Floating point (double)
- Document (document)
- Array (document)
- Binary data (binary)
- Boolean false (`\x00` or byte 0000 0000)
- Boolean true (`\x01` or byte 0000 0001)
- UTC datetime (int64)—the int64 is UTC milliseconds since the Unix epoch



# BSON Data Types II


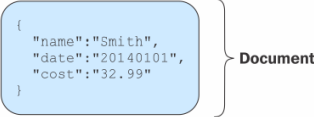
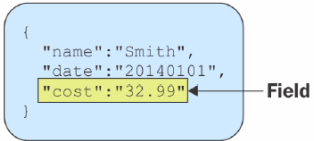
- Timestamp (int64)—this is the special internal type used by MongoDB replication and sharding; the first 4 bytes are an increment, and the last 4 are a timestamp
- Null value ()
- Regular expression (cstring)
- JavaScript code (string)
- JavaScript code w/scope (code\_w\_s)
- Min key()—the special type that compares a lower value than all other possible BSON element values
- Max key()—the special type that compares a higher value than all other possible BSON element values
- ObjectId (byte\*12)



In order to model data in MongoDB, some constraints must be satisfied:

- Maximum length for a BSON document is 16 MB, for bigger sizes a GridFS must be used
- The `_id` field is reserved for a primary key
- Besides default ObjectID `_id`, any value can be used as long it is unique (except arrays)
- A name cannot start with the character `$`
- A name cannot have a null character, or `(.)`

# Documents, collections, and database

MongoDB	RDBMS	Example
Collection	Table	
Document	Row	
Field	Column	



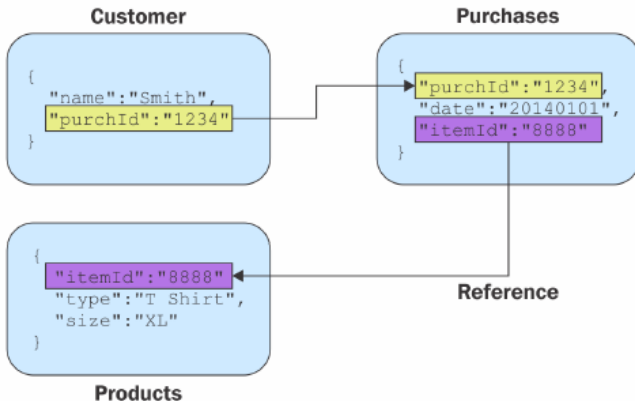
# Dealing With Redundancy

Normalisation is a fundamental process to help build relational data models: from the basic 1NF (first normal form), onto the 2NF, 3NF, and BCNF

To minimise redundancy, we divide larger tables into smaller ones and define relationships among them

Creating a reference in MongoDB is a way to "normalise" our model and represent relationships between documents

Using references (DBRefs), it is possible to create a series of related collections in order to establish a normalised data model.





# JOINS do exist in MongoDB ?

Forcing a highly-normalized solution on a MongoDB dataset, basically defeat the purpose of using a NoSQL database.

MongoDB DOES NOT support joins, while using references, you must perform at least two queries to get the whole information

## DBRefs ARE NOT like foreign keys in RDBMS

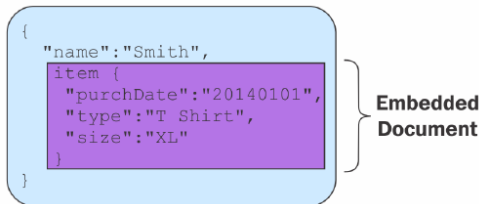
If your database driver didn't provide support for DBRefs, you are forced to write code to traverse the references manually.

This introduces the very overhead you wanted to avoid by choosing MongoDB in the first place!

Example: delete a document pointed from another document ...  
MongoDB is fine, your applicationn is NOT !!!

A suitable NoSQL data-modelling solution would be simply to collapse the normalised relationships and fold the related information into embedded documents.

With respect to the previous example, the solution would appear as follows:







Modeling application data for MongoDB should consider various operational factors

Different data models can allow for more efficient queries, increase the throughput of insert and update operations, or distribute activity to a sharded cluster more effectively

## Hint

When developing a data model, there is the need to analyze all of application's read and write operations in conjunction with atomicity, sharding, indexing, large number of collections



- RDBMS are guaranteeing ACID (Atomicity, Consistency, Isolation, and Durability) properties
- In case of distributed datasets it holds the CAP (Consistency, Availability and Partition Tolerance) theorem
- BASE (Basically Available, Soft state, Eventual consistency)



A write operation is atomic on the level of a single document in MongoDB, even if the operation involves embedded documents within

If atomicity is required for updates to multiple documents (or consistency between reads to multiple documents), MongoDB provides multi-document transactions in an “all-or-nothing” proposition from version 4.0

Multi-document transactions are available for replica sets only, while for sharded clusters are scheduled for MongoDB 4.2



Designing a schema for managing data using a document must be aware of:

- Whether consistency is the priority
- Whether read is the priority
- Whether write is the priority
- What update queries we will make
- Document growth



# One To One - Referenced Data

```
customer
{
  "_id": 5478329cb9617c750245893b
  "username": "John Clay",
  "email": "johnclay@crgv.com",
  "password": "bf383e8469e98b44895d61b821748ae1"
}
customerDetails
{
  "customer_id": "5478329cb9617c750245893b",
  "firstName": "John",
  "lastName": "Clay",
  "gender": "male",
  "age": 25
}
```



```
customer
{
  _id: 1
  "username" : "John Clay",
  "email": "johnclay@crgv.com",
  "password": "bf383e8469e98b44895d61b821748ae1"
  "details": {
    "firstName": "John",
    "lastName": "Clay",
    "gender": "male",
    "age": 25
  }
}
```



# One To Many - Referenced Data

```
customer
{
  _id: 1
  "username" : "John Clay",
  "email": "johnclay@crgv.com",
  "password": "bf383e8469e98b44895d61b821748ae1"
  "details": {
    "firstName": "John",
    "lastName": "Clay",
    "age": 25 }
}

address
{
  _id: 1,
  "street": "Address 1, 111",
  "city": "City One",
  "type": "billing",
  "customer_id": 1
}
{
  _id: 2,
  "street": "Address 2, 222",
  "city": "City Two",
  "type": "shipping",
  "customer_id": 1
}
{
  _id: 3,
  "street": "Address 3, 333",
  "city": "City Three",
  "type": "shipping",
  "customer_id": 1
}
```



# One To Many - Embedded Data

```
customer
{
  _id: 1
  "username" : "John Clay",
  "email": "johnclay@crgv.com",
  "password": "bf383e8469e98b44895d61b821748ae1"
  "details": {
    "firstName": "John",
    "lastName": "Clay",
    "gender": "male",
    "age": 25
  }
  "billingAddress": [{
    "street": "Address 1, 111",
    "city": "City One",
    "type": "billing",
  }],
  "shippingAddress": [{
    "street": "Address 2, 222",
    "city": "City Two",
    "type": "shipping"
  }, {
    "street": "Address 3, 333",
    "city": "City Three",
    "type": "shipping"
  }]
}
```





In the relational model, this kind of relationship is often represented as a join table while, in the non-relational one, it can be represented in many different ways

```
user
{
  _id: "5477fdea8ed5881af6541bf1",
  "username": "user_1",
  "password" : "3f49044c1469c6990a665f46ec6c0a41"
}

{
  _id: "54781c7708917e552d794c59",
  "username": "user_2",
  "password" : "15e1576abc700ddfd9438e6ad1c86100"
}

group
{
  _id: "54781cae13a6c93f67bdcc0a",
  "name": "group_1"
}

{
  _id: "54781d4378573ed5c2ce6100",
  "name": "group_2"
}
```



# Many To Many - Embed Data (1)

```
user
{
  (unchanged)
}
group
{
  _id: "54781cae13a6c93f67bdcc0a",
  "name": "group_1",
  "users": [{
    _id: "54781c7708917e552d794c59",
    "username": "user_2",
    "password": "15e1576abc700ddfd9438e6ad1c86100"
  }]
}
{
  _id: "54781d4378573ed5c2ce6100",
  "name": "group_2",
  "users": [{
    _id: "5477fdea8ed5881af6541bf1",
    "username": "user_1",
    "password": "3f49044c1469c6990a665f46ec6c0a41"
  }, {
    _id: "54781c7708917e552d794c59",
    "username": "user_2",
    "password": "15e1576abc700ddfd9438e6ad1c86100"
  }]
}
```



# Many To Many - Embed Data (2)

```
user
{
  _id: "5477fdea8ed5881af6541bf1",
  "username": "user_1",
  "password": "3f49044c1469c6990a665f46ec6c0a41",
  "groups": [
    {
      _id: "54781cae13a6c93f67bdcc0a",
      "name": "group_1"
    },
    {
      _id: "54781d4378573ed5c2ce6100",
      "name": "group_2"
    }
  ]
}
{
  _id: "54781c7708917e552d794c59",
  "username": "user_2",
  "password": "15e1576abc700ddfd9438e6ad1c86100",
  "groups": [
    {
      _id: "54781d4378573ed5c2ce6100",
      "name": "group_2"
    }
  ]
}
group
{
  (unchanged)
}
```



# Many To Many - Embed Data (3)

```
user
{
  _id: "5477fdea8ed5881af6541bf1",
  "username": "user_1",
  "password": "3f49044c1469c6990a665f46ec6c0a41",
  "groups": ["54781cae13a6c93f67bdcc0a", "54781d4378573ed5c2ce6100"]
}
{
  _id: "54781c7708917e552d794c59",
  "username": "user_2",
  "password": "15e1576abc700ddfd9438e6ad1c86100",
  "groups": ["54781d4378573ed5c2ce6100"]
}

group
{
  _id: "54781cae13a6c93f67bdcc0a",
  "name": "group_1",
  "users": ["5477fdea8ed5881af6541bf1"]
}
{
  _id: "54781d4378573ed5c2ce6100",
  "name": "group_2",
  "users": ["5477fdea8ed5881af6541bf1", "54781c7708917e552d794c59"]
}
```

# MongoDB Indexes



Indexes improve MongoDB performance at the collection level avoiding full scan

Indexes in MongoDB are similar to indexes in other database systems and are defined at the collection level objects

There are three types of indexes in MongoDB: *single field*, *compound*, *multi-key*, each can be *ascending* or *descending*

An automated index on *\_id* field is generated by default on every collection.



# Unique Indexes

In the RDBMS, it is a *good practice* to define one or more columns as the primary key in order to identify a given row uniquely.

In MongoDB a unique identifying field `_id` is *automatically* inserting a document into a collection.

This field comprises an ObjectId instance and uses a combination of factors to guarantee uniqueness.



ObjectId are likely unique, fast to generate, and ordered.

ObjectId values consist of 12 bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation:

- a 4-byte value representing the seconds since the Unix epoch
- a 5-byte random value
- a 3-byte counter, starting with a random value





# Unique Indexes - Custom Default Index

In MongoDB the default identifying field `_id` is an `ObjectId`:

`_id` as `ObjectId`

```
{ "_id": ObjectId("570c04a4ad233577f97dc459"),  
  "temperature": [ 22, 12, 30 ] }
```

Anyway, it is possible to set a custom default index:

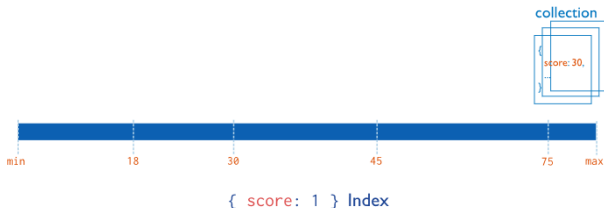
`_id` as integer

```
{ "_id": 20180106,  
  "temperature": [ 22, 12, 30 ] }
```



# Single Field Indexes

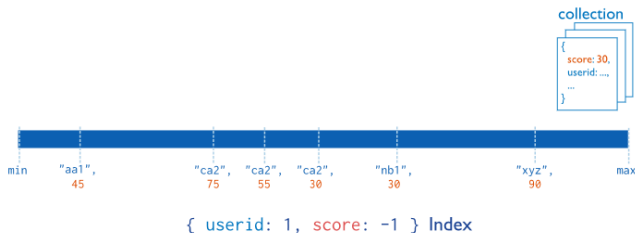
Single field indexes speed up operations that involve a particular (single) field



## Tip

For single-field index and sort operations, the sort order (i.e. ascending or descending) does not matter because MongoDB can traverse the index in either direction

Compound indexes are useful to index more than one field



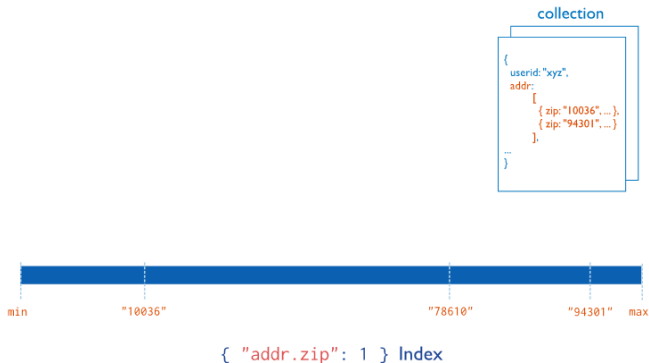
## Warning

For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support (or not) a sort operation.



# Multi-key Indexes

Multikey indexes are for content stored in array, separate index entries are created for every element of the array





- A text index of a string or an array of string fields can be created in a collection
- Only one text index per collection can exist
- It is possible to create a compound text index
- Index build process can be split it into three phases:
  - Tokenization
  - Removal of suffix and/or prefix, or stemming
  - Removal of stop words



# Wildcard Text Indexes

The wildcard specifier (\$\*\*) creates a text index on multiple fields, MongoDB indexes *every field that contains string data* for each document in the collection

Wildcard text indexes are text indexes on multiple fields, it is possible to assign weights to specific fields during index creation to control the ranking of the results

## Hint

This index allows for text search on all fields with string content, it can be useful with highly unstructured data if it is unclear which fields to include in the text index or for ad-hoc querying



# Time To Live Indexes

- The time to live (TTL) index is an index based on lifetime
- They cannot be compound and they will be automatically removed from the document after a given period of time
- MongoDB is responsible for controlling the documents' expiration through a background task at intervals of 60 seconds

## Hint

Useful for applications that use machine-generated events, logs and session information, which need to be persistent only during a given period of time

# Replication



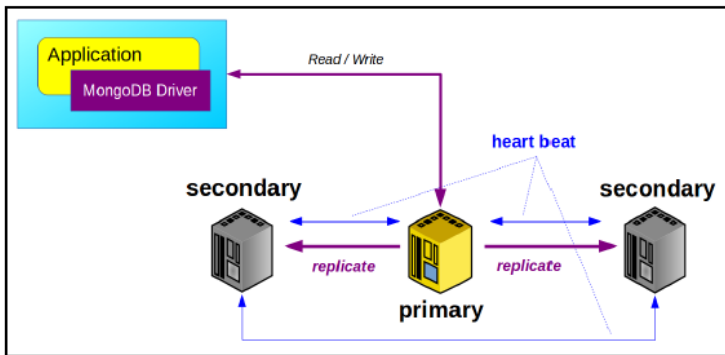


Replication is needed to provide immediate redundancy to avoid loss of services in case of failure

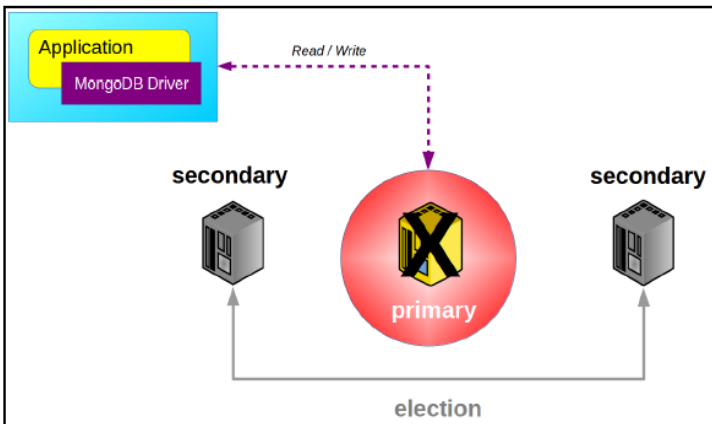
A *replica set* is one or more MongoDB daemon instances (or nodes), with the same data.

One of the nodes is elected to *intercept all reads and writes*. This node is referred to as the *primary*

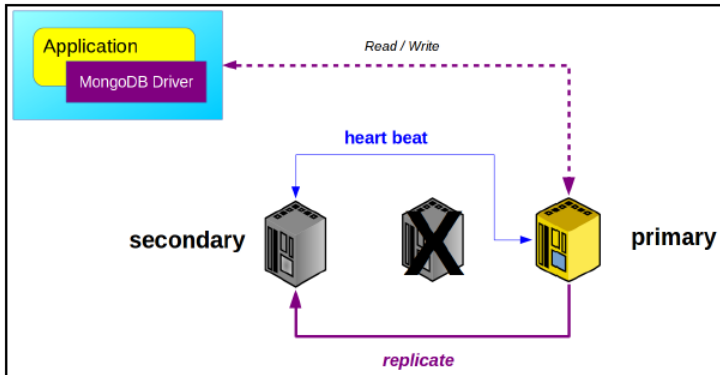
# Replication Example - 1



# Replication Example - 2



# Replication Example - 3



# Sharding

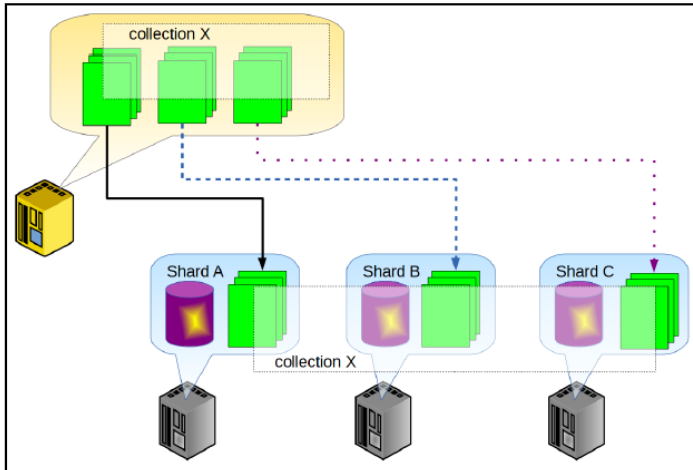


Replication provides redundancy of data while sharding provides horizontal scaling

Instead of a single, very powerful, server (vertical scaling), data is spread across a cluster made with multiple servers (cloud environment)

Each server is then able to handle its own database tasks, in this way it is possible to handle massive amounts of data

# Sharding Example





Sharding in MongoDB operates at the *collection* level by partitioning it into several *chunks* of configurable size (up to 64 Mb)

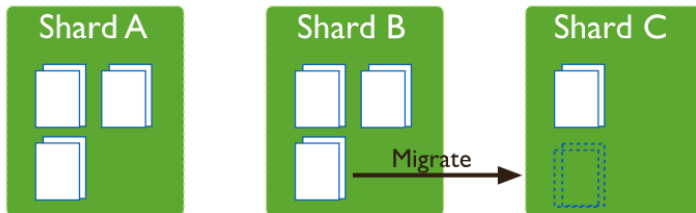
Chunks are distributed across the shards on the basis of the *Shard Key*

A database can have a mixture of sharded and unsharded collections



MongoDB *balancer* is a (transparent) process monitoring the number of chunks on each shard

If the number of chunks on a given shard reaches a thresholds, chunks are automatically migrated between shards to reach an equal number of chunks per shard.





# Choosing a Shard Key

The shard key is a document field(s) used by MongoDB to distribute documents between the shards

The chosen field must be:

- immutable (cannot be updated once written)
- must exist in *every* single document
- unique within a collection (one and only)
- once selected it cannot be substitute by another field

## Important

A sharded collection can grow to any size after successful sharding (adding necessari nodes)



# Shard Key Criteria

Shard keys are chosen based on the following criteria:

- *Cardinality*: Number of elements in a set (maximum number of chunks)
- *Frequency*: How often a given value occurs in the data (low frequency equals low distribution)
- *Monotonically Changing* : How the elements in the set are added to or removed from this set (avoid outermost-only operations)

## Hint

It is possible to create a new field which could be a combination of some other field and use it as shard key.



Basically MongoDB offers three sharding strategies:

- *Ranged sharding*
- *Hashed sharding*
- *Zoned Sharding*

The choice of strategy depends heavily on the shard key chosen and it will allow MongoDB to distribute documents evenly throughout the sharded cluster



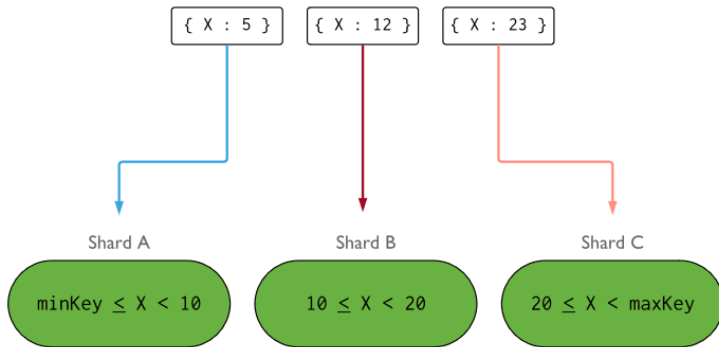
Ranged sharding strategy, the default, distributes documents into chunks based on the value of the shard key. The chunks are then automatically distributed across the sharded cluster by the balancer.

Ranged sharding is recommended if the shard key has the following characteristics:

- High cardinality
- Low Frequency
- Low Monotonically Changing



# Ranged Sharding Example





Hashed Sharding ensures an even distribution of data across shards, but by losing the ability to target specific shards knowing the contents of the shard key field.

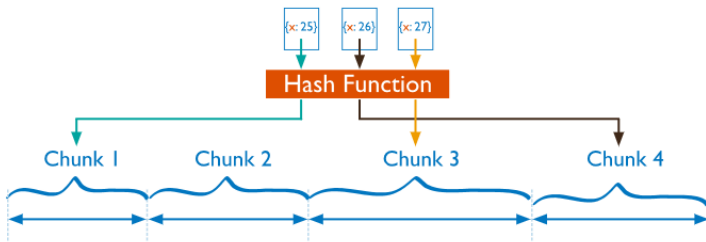
Use hashed sharding if your shard key has either of the following characteristics:

- Low Cardinality
- High Monotonically Changing



# Hashed Sharding Example

Hashed sharding uses a hashed index to partition data across shared cluster, documents with “close” shard key values are unlikely to be on the same chunk or shard



## Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes





In sharded clusters zones of sharded data based on the shard key can be created.

A zone can be associated with one or more shards in the cluster, and shard can associate with any number of zones.

Applications:

- isolate a specific subset of data on a specific set of shards.
- ensure that the most relevant data reside on shards that are geographically closest to the application servers.
- route data to shards based on the hardware/performance of the shard hardware.

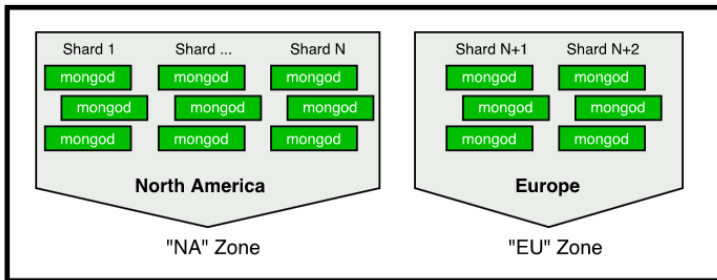


# Zones: Segmenting Data by Location

Example use cases for segmenting data by geographic area:

- An application requires segmenting user data based on geographic country
- A database requires resource allocation based on geographic country

## Sharded Cluster





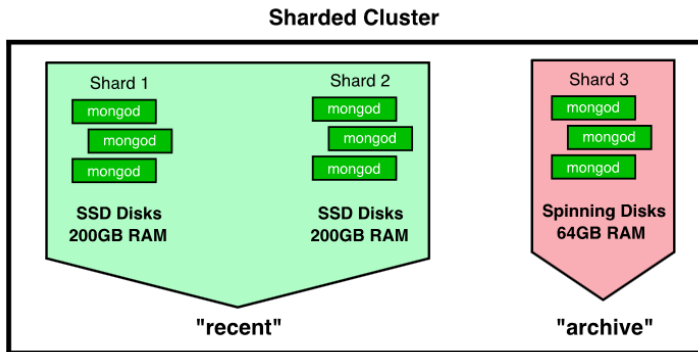
# Zones: Tiered HW for Varying SLA or SLO

Examples of use cases for segmenting data based on Service Level Agreement (SLA) or Service Level Objective (SLO):

- providing low-latency access to recently inserted / updated documents
- providing low-latency access to recently inserted / updated documents
- ensuring specific ranges or subsets of data are stored on servers with hardware that suits the SLA's for accessing that data

# Zones for SLA Example

An example of photo sharing application requiring fast access to photos uploaded within the last 6 months



# Aggregation and Mapreduce



Aggregation operations process data records and return computed results.

Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

MongoDB provides three ways to perform aggregation:

- aggregation pipeline
- map-reduce function
- single purpose aggregation methods



# Aggregation Pipeline

Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

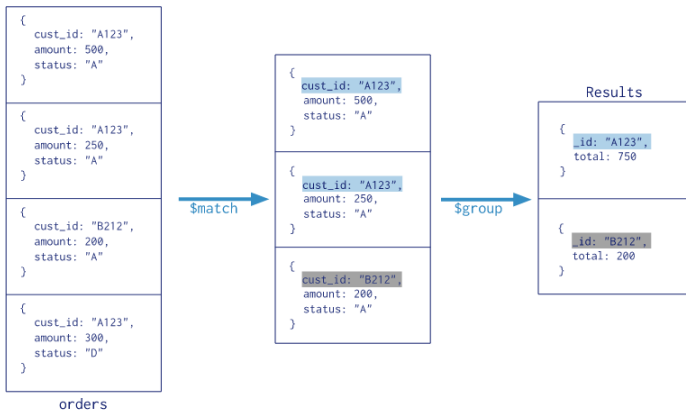
The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document

The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase.



# Aggregation Pipeline Example

Collection  
↓  
db.orders.aggregate( [  
 \$match stage → { \$match: { status: "A" } },  
 \$group stage → { \$group: { \_id: "\$cust\_id", total: { \$sum: "\$amount" } } }  
] )







MongoDB also provides map-reduce operations to perform aggregation.

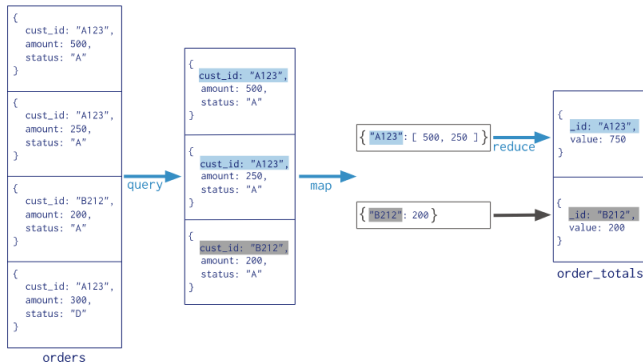
Map-reduce operations have two phases:

- a map stage that processes each document and emits one or more objects for each input document
- and reduce phase that combines the output of the map operation

Map-reduce can operate on a sharded collection. Map-reduce operations can also output to a sharded collection.

# Map-Reduce Example

Collection  
↓  
db.orders.mapReduce(  
  map    → function() { emit( this.cust\_id, this.amount ); },  
  reduce → function(key, values) { return Array.sum( values ) },  
  
  query → { query: { status: "A" } },  
  output → { out: "order\_totals" }  
)





Doug Bierer (2018). *MongoDB 4 Quick Start Guide*. Packt Publishing Ltd.

Alex Giamas (2017). *Mastering MongoDB 3.x*. Apress.

*MongoDB Architecture Guide* (2018). MongoDB, Inc.

Wilson da Rocha França (2015). *MongoDB Data Modeling*. Packt Publishing Ltd.

*The MongoDB 4.0 Manual* (2018). MongoDB, Inc.