Indexing

Indexing is done for the faster retrial of data. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement.

Type of Index:

Default_id – every collection contains an index on default _id field.

Single Field – An index created on a single field in ascending or descending order.

Compound Index – An index based on multiple fields (max 31)

Multi-Key Index – index created on array field

TTL Index – TTL (Total Time to Live) are created for a limited time

Unique Index – ensures the uniqueness of the field

Indexing

Create index – db.collectionname.createIndex()

Single Field – creating a single field index in ascending order on city field in emp3 collection.

```
> db.emp3.createIndex({"City" : 1}) ;
```

Compound Index – creating a compound index in ascending order on EID & City field in emp3 collection.

```
> db.emp3.createIndex({"EID" : 1, "City" : 1});
```

Indexing

TTL Index – TTL (Total Time to Live) are created by combining “expireAfterSeconds” and createIndex().

```
> db.emp3.createIndex({"EID" : 1},{expireAfterSeconds:600});
```

TTL Limitations:

Compound Indexes can not be created as TTL

Can not be created on capped collections

Cannot be created on the field on which other index exists

Indexing

Unique Index – to create a unique index we need to set the unique option of createIndex() method to true

```
db.collectionname.createIndex({key: 1},{unique:true})
```

```
>db.emp3.createIndex({"EID" : 1},{unique:true});
```

A unique index allows 1 null value.

Indexing

Other Index methods

Find index – db.collectionname.getIndexes()

Drop index – db.collectionname.dropIndex()

Drop All index - db.collectionname.dropIndexes();

```
> db.emp.getIndexes();
> db.emp.dropIndex({"EID" : 1});
> db.emp.dropIndexes();
```

Note: A collection can have maximum 64 indexes.

Aggregating Documents

Aggregations operations process documents 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. A SQL function (sum()) along with group by is an equivalent to MongoDB aggregation.

```
>db.COLLECTION_NAME.aggregate([AGGREGATE_OPERATION])
```

Aggregating Documents

Below are the few aggregation expressions:

$sum - Sums up the defined value from all documents in the collection

$avg - Calculates the average of all given values from all documents in the collection

$min - Gets the minimum of the corresponding values from all documents in the collection

$max - Gets the maximum of the corresponding values from all documents in the collection

Aggregating Documents

Below aggregate method will give total cost for each department.

```
> db.salary.aggregate(
                        [{$group :
                                {_id : "$DEPT",
                                        TotalCost :{$sum :  "$SALARY"}
                                }
                        }]
                        );
```

```
> db.salary.aggregate([{$group: {_id : "$DEPT" , "teamsize" : {$sum : 1}}}]);
```

```
> db.salary.aggregate([{$group: {_id : "$DEPT" , "teamsize" : {$sum : 1},"cost" :
  vg: "$SALARY"}}}]);
```

Aggregating Documents

Below aggregate method will give total, avg, maximum & minimum salary for each department.

```
> db.salary.aggregate(
                        [{$group :
                                {_id : "$DEPT",
                                        TotalCost :{$sum :  "$SALARY"},
                                        Avgsal :{$avg :  "$SALARY"},
                                        Minsal :{$min :  "$SALARY"},
                                        Maxsal :{$max :  "$SALARY"}
                                }
                        }]
                        );
```

Atomic Operations

To maintain atomicity it is recommended to keep all the related information which is updated together in a single embedded Document.

Example:

```
>db.order.Insert(
                {"_id" : 1,
                 "pdesc" : "Dell Mouse",
                "category" :  "IT" ,
                "Totalstk": 10,
                 "balanceStk" : 8,
                “purchasedby" : [
                                {"cname" : "ajay kumar", "date" : "1-May-2021”},
                                {"cname" : "ravi sharma", "date" : "2-May-2021”}
                                ]
                }
                );
```

Atomic Operations

In this example we want when ever the customer order the product, the availability will be checked, if the stock is available, the balance should be reduced and the customer information should be added in the document.

```
> db.order.findAndModify({
                    query: {"_id":1,"balanceStk" :{$gt : 0}},
                    update: {
                            $inc: {"balanceStk" : -1},
                            $push: {"boughtby":
                            {"cname":"Gaurav","date" : Date()}}
                            }
                    });
```

Atomic Operations

findAndModify() – search for the document and modify it.

query: {} – specify the search criteria

update: {} – specify the updation in the document

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

$push: - Adds an item to an array.

- In the employee collection create an unique index on EID field.
- Create a TTL index on City for 10 min.
- List 5 highest paid employees from sal collection.
- Total Salary, Team Size & Average salary for each department.
- Create a embedded document containing product information & an array for customer information. Perform a operation to add the customer info in to an array and update the available stock.

Data Modeling with MongoDB

RDBMS approach Vs MongoDb

RDBMS:

1. Define the normalized schema
2. Develop the application & queries – the application is designed as per the data. Concerns ?Usage ?Performance

MongoDb:

1. Develop the application
2. Define the data model
3. Improve the application
4. Improve the Data Model - No down time, designed for usage pattern, Data model evolution is easy

Data Modeling with MongoDB

Data model is designed at the application level.

Design is the part of each phase of the application lifetime

The data which the application needs and read write usage of data affects the data model (more read/more write optimised for the purpose)

The data Model will Evolve

Data Model in MongoDB

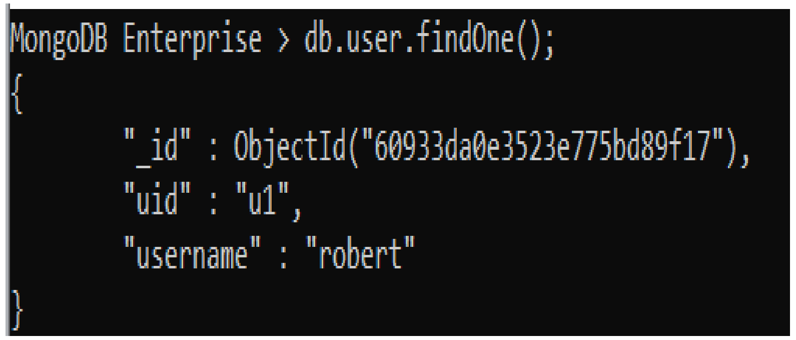
Normalized data model : In this model every sub document has an id. Normalized data models describe relationships using references between documents.

In general, use normalized data models:

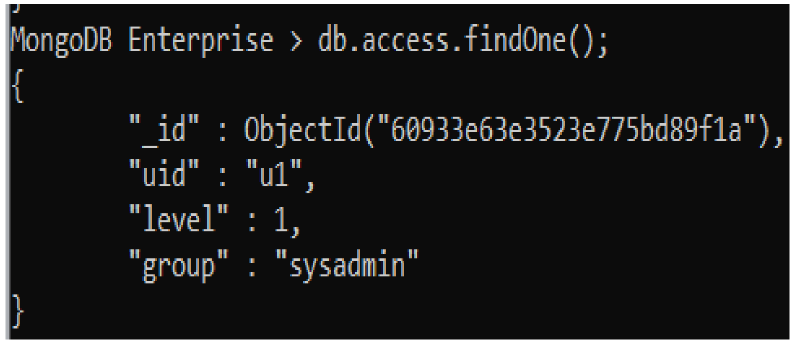
- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

Data Model in MongoDB

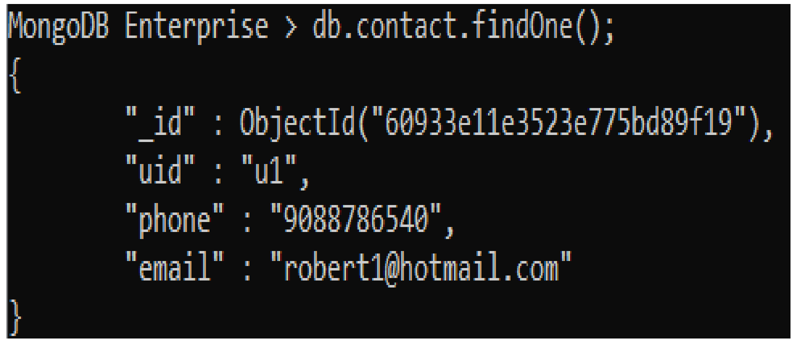
Normalized data model



```
MongoDB Enterprise > db.user.findOne();
{
        "_id" : ObjectId("60933da0e3523e775bd89f17"),
        "uid" : "u1",
        "username" : "robert"
}
```



```
MongoDB Enterprise > db.access.findOne();
{
        "_id" : ObjectId("60933e63e3523e775bd89f1a"),
        "uid" : "u1",
        "level" : 1,
        "group" : "sysadmin"
}
```



```
MongoDB Enterprise > db.contact.findOne();
{
        "_id" : ObjectId("60933e11e3523e775bd89f19"),
        "uid" : "u1",
        "phone" : "9088786540",
        "email" : "robert1@hotmail.com"
}
```

Data Model in MongoDB

Embedded data model :In this model, all the related data is in a single document, it is also known as de-normalized data model. In this model one document can be embedded as the sub document in the other document. Sub document can be stored as array or JSON object.

```
db.emp2.insertOne({eid: "e0001",
                            PD: {fn : "Amit", ln : "kumar", dob : "10-may-1990"},
                            contact: {ph : "98899787690", email :
"akumar@gmail.com"},
                            addr: {area : "sector 5 Dwarka", city : "delhi"},
                            off: {dept : "ops", desi : "manager" , salary : 90000}
                   });
```

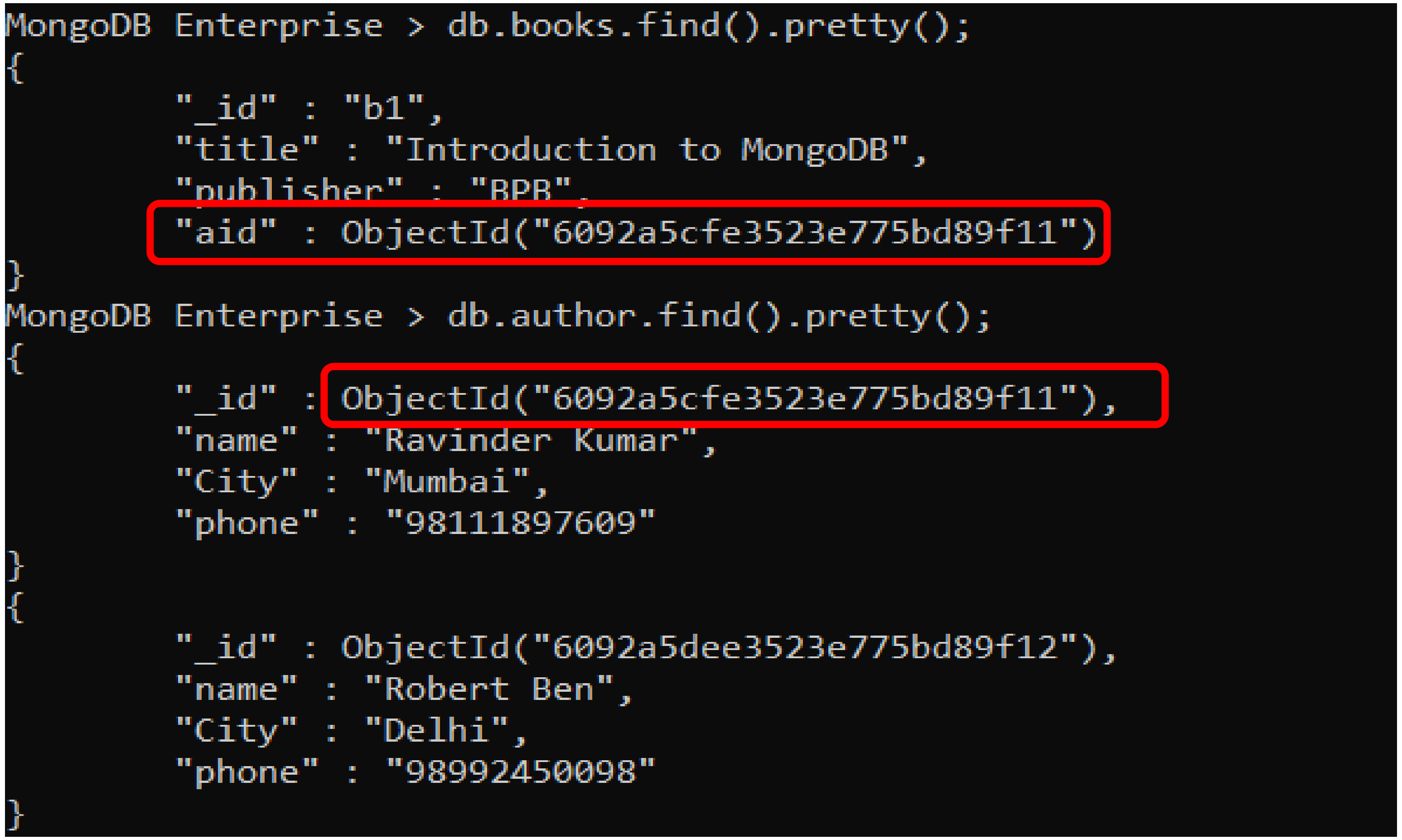
Data Model in MongoDB

De-Normalized data model

```
db.emp2.insertOne({_id : "OID101" , eid: "e0002",
                              PD: {_id : "OID102" , empDocID: "OID101", fn : "kapil", ln :
"sharma", dob : "10-may-1990"},
                              contact: {_id : "OID103" , empDocID: "OID101", ph :
"98899000888", email : "kapil@gmail.com"},
                              addr: {_id : "OID104" , empDocID: "OID101", area : "sector 3
rohini", city : "delhi"},
                              off: {_id : "OID105" , empDocID: "OID101", dept : "ops", desi :
"associate" , salary : 40000}
                    }
);
```

Data Model in MongoDB

Referencing: Inserting the object Id of one document in another document is known as referencing.



```
MongoDB Enterprise > db.books.find().pretty();
{
        "_id" : "b1",
        "title" : "Introduction to MongoDB",
        "publisher" : "BPB",
        "aid" : ObjectId("6092a5cfe3523e775bd89f11")
}
MongoDB Enterprise > db.author.find().pretty();
{
        "_id" : ObjectId("6092a5cfe3523e775bd89f11"),
        "name" : "Ravinder Kumar",
        "City" : "Mumbai",
        "phone" : "98111897609"
}
{
        "_id" : ObjectId("6092a5dee3523e775bd89f12"),
        "name" : "Robert Ben",
        "City" : "Delhi",
        "phone" : "98992450098"
}
```

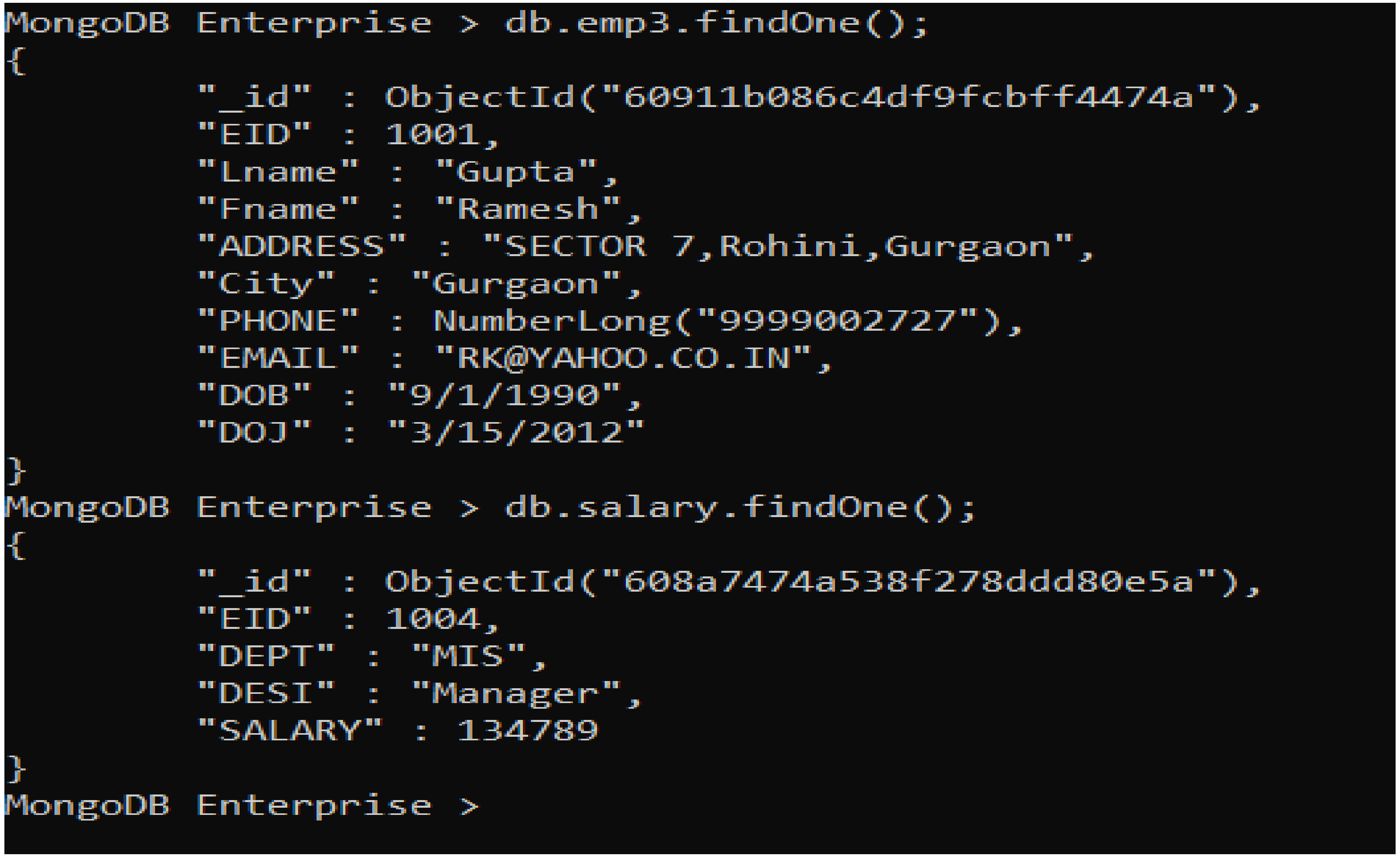
Data Model in MongoDB

<u>$lookup (aggregation)</u>: it adds an array of related data from the other document. It perform an equality match between a field from the input documents with a field from the documents of the "joined" collection.

```
>db.collectionname.aggregate({

   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
 });
```

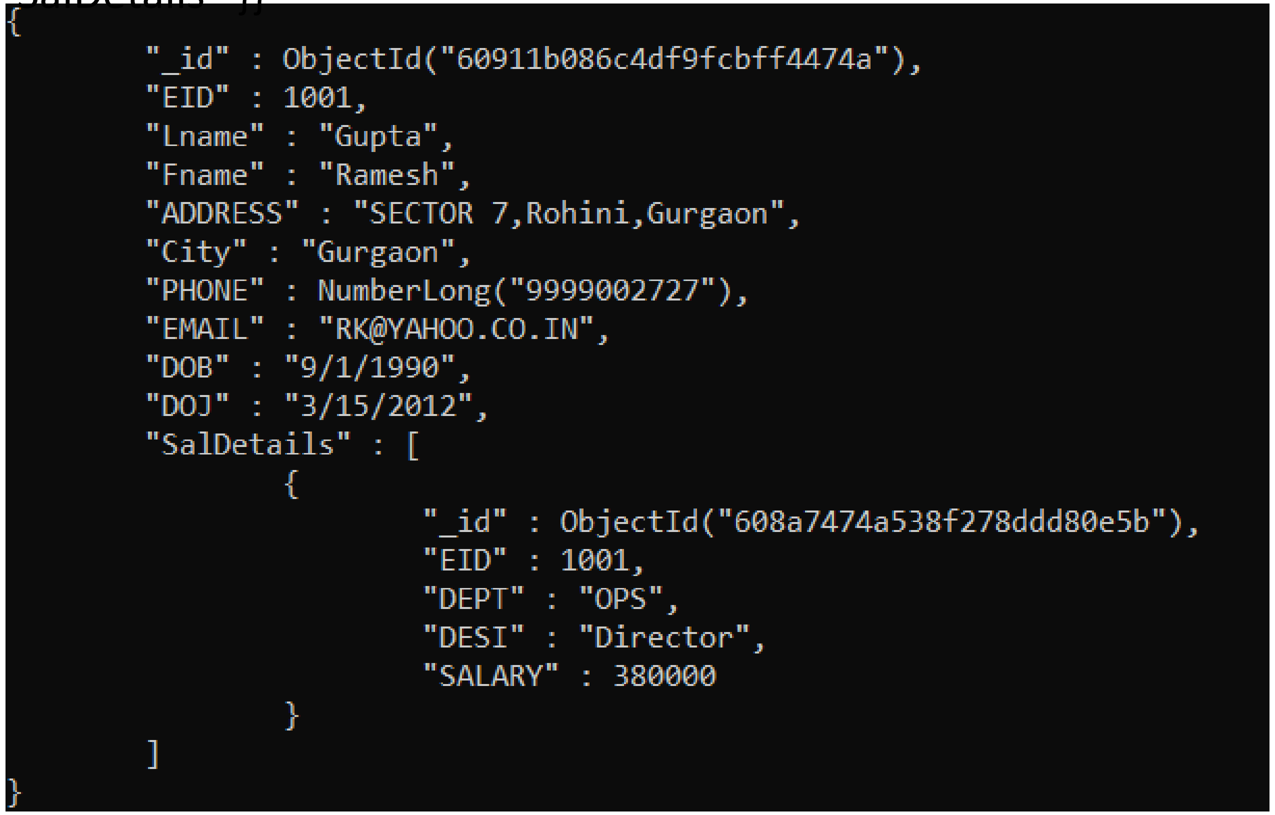
Data Model in MongoDB



```
MongoDB Enterprise > db.emp3.findOne();
{
        "_id" : ObjectId("60911b086c4df9fcbff4474a"),
        "EID" : 1001,
        "Lname" : "Gupta",
        "Fname" : "Ramesh",
        "ADDRESS" : "SECTOR 7,Rohini,Gurgaon",
        "City" : "Gurgaon",
        "PHONE" : NumberLong("9999002727"),
        "EMAIL" : "RK@YAHOO.CO.IN",
        "DOB" : "9/1/1990",
        "DOJ" : "3/15/2012"
}
MongoDB Enterprise > db.salary.findOne();
{
        "_id" : ObjectId("608a7474a538f278ddd80e5a"),
        "EID" : 1004,
        "DEPT" : "MIS",
        "DESI" : "Manager",
        "SALARY" : 134789
}
MongoDB Enterprise >
```

Data Model in MongoDB

```
> db.emp3.aggregate([
            {$lookup:
            {from: "salary" ,
            localField: "EID",
            foreignField: "EID",
            as: "SalDetails" }}
            ]);
```

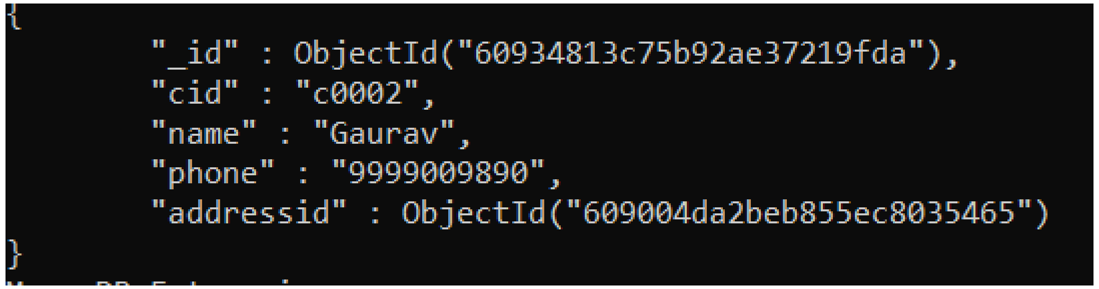


```
{
        "_id" : ObjectId("60911b086c4df9fcbff4474a"),
        "EID" : 1001,
        "Lname" : "Gupta",
        "Fname" : "Ramesh",
        "ADDRESS" : "SECTOR 7,Rohini,Gurgaon",
        "City" : "Gurgaon",
        "PHONE" : NumberLong("9999002727"),
        "EMAIL" : "RK@YAHOO.CO.IN",
        "DOB" : "9/1/1990",
        "DOJ" : "3/15/2012",
        "SalDetails" : [
                {
                        "_id" : ObjectId("608a7474a538f278ddd80e5b"),
                        "EID" : 1001,
                        "DEPT" : "OPS",
                        "DESI" : "Director",
                        "SALARY" : 380000
                }
        ]
}
```

Relationship in MongoDB

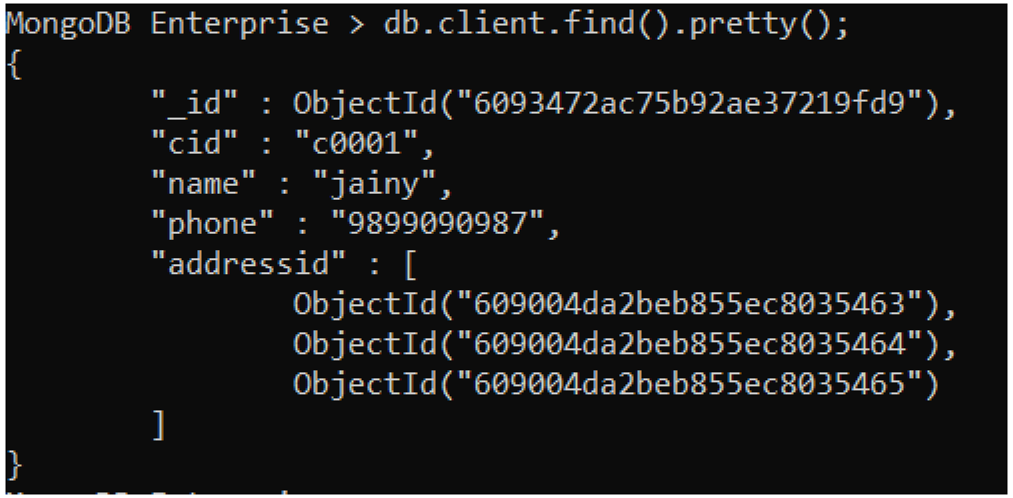
Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via Embedded and Referenced approaches. Such relationships can be either

One – to - One (1:1),



```
{
        "_id" : ObjectId("60934813c75b92ae37219fda"),
        "cid" : "c0002",
        "name" : "Gaurav",
        "phone" : "9999009890",
        "addressid" : ObjectId("609004da2beb855ec8035465")
}
```

One – to – Many(1:N),



```
MongoDB Enterprise > db.client.find().pretty();
{
        "_id" : ObjectId("6093472ac75b92ae37219fd9"),
        "cid" : "c0001",
        "name" : "jainy",
        "phone" : "9899090987",
        "addressid" : [
                ObjectId("609004da2beb855ec8035463"),
                ObjectId("609004da2beb855ec8035464"),
                ObjectId("609004da2beb855ec8035465")
        ]
}
```

Many – to- One (N:1)

Many – to- Many (N:N)

MongoDb Drivers

Drivers are the packages we install for different programming languages in which the application might be written.

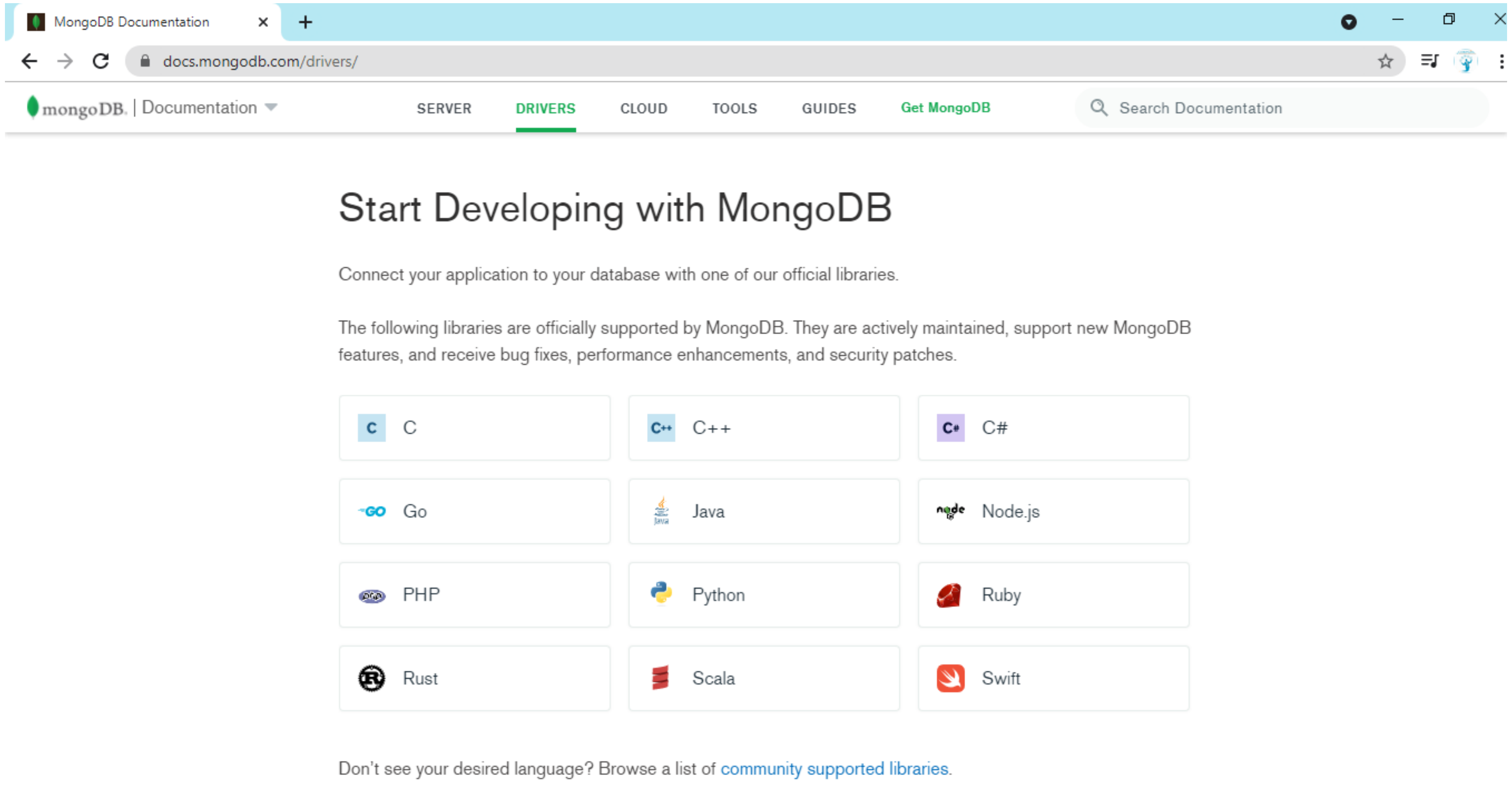
These can be downloaded for MongoDb official page under the doc tab:

[Start Developing with MongoDB — MongoDB Drivers]()

These are the bridge between the database & the application

MongoDb Drivers

Choose the appropriate driver as per your application



Start Developing with MongoDB

Connect your application to your database with one of our official libraries.

The following libraries are officially supported by MongoDB. They are actively maintained, support new MongoDB features, and receive bug fixes, performance enhancements, and security patches.

C	C++	C#
Go	Java	Node.js
PHP	Python	Ruby
Rust	Scala	Swift

Don't see your desired language? Browse a list of community supported libraries.

ASSIGNMENT

- Create a document to track book details and author details should be added as a reference document.
- Retrieve the book title ,publisher & and author name using $lookup aggregation.
- Display the aggregated data of employee and salary details.

Data Types in MongoDB

In MongoDB data is data representation is done in JSON (JavaScript Object Notation) document format which is binary encoded and is termed as BSON. MongoDB supports many data types. Such as :

Integer − This type is used to store a numerical value.

```
> db.testdt.insert({"integer" :125});
```

Boolean − This type is used to store a boolean (true/ false) value.

```
> db.testdt.insert({"registered" :true});
```

Double − This type is used to store floating point values.

```
> db.testdt.insert({"amount" : 3745.95});
```

String − This is the most commonly used datatype to store the data.

```
> db.testdt.insert({“greeting" : “Welcome to MongoDB”});
```

Arrays − This type is used to store arrays or list or multiple values into one key.

```
> var courses = ["SQL" ,"PBI", "MongoDB"]
> db.testdt.insert({"module" : courses});
```

Data Typesin MongoDB

Object − This datatype is used for embedded documents.

```
> var hrs = {"SQL" : 25, "PowerBi" :20 , "MongoDB" : 15};
> db.testdt.insert({"Duration" : hrs});
```

Null − This type is used to store a Null value.

```
> db.testdt.insert({"email" : null});
```

Date − This datatype is used to store the date.

```
> var d1 = Date();
> var d2 = ISODate();
> var JD = ISODate("2021-05-01");
> db.testdt.insert({"Stringdate" : d1, "ISODate" : d2 , "JoiningDate": JD});
```

Timestamp − Timestamp stores 64-bit value. This can be handy for recording when a document has been modified or added.

```
>var v1= new Timestamp();
> db.testdt.insert({"login": v1 });
```

Data Types in MongoDB

Object ID – This datatype is used to store the document's ID.

```
> var v1 = ObjectId("6093b6d573c5517d62544e4e");
> db.testdt.insert({"refdocid" : v1});
```

MongoDb also allows us to query the data on the basis of data type.

$type selects documents where the value of the field is an instance of the specified type. Querying by data type is useful when dealing with highly unstructured data where data types are not predictable.

```
{ field: { $type: <BSON type> } }
```

```
> db.testdt.find({"JoiningDate" : {$type : "date"}})
> db.testdt.find({"JoiningDate" : {$type : "string"}})
```