Sorting Documents

sort() method is used to sort the documents. The method accepts a document containing a list of fields along with their sorting order. 1 is used for ascending order while -1 is used for descending order.

```
>db.COLLECTION_NAME.find().sort({KEY:1})
```

```
> db.salary.find().sort({“EID" :1});
> db.salary.find({}).limit(5).sort({"SALARY" : 1});
> db.salary.find({},{"EID" : 1 , "DESI" : 1}).limit(5).sort({"EID" : 1})
> db.emp3.find({},{"EID" : 1, "Fname" :1 ,"City" :1}).sort({"City" :
1,"EID" :1}).limit(10);
```

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" : {$sum : 1},"cost" :
{$avg: "$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 emp document.
- 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

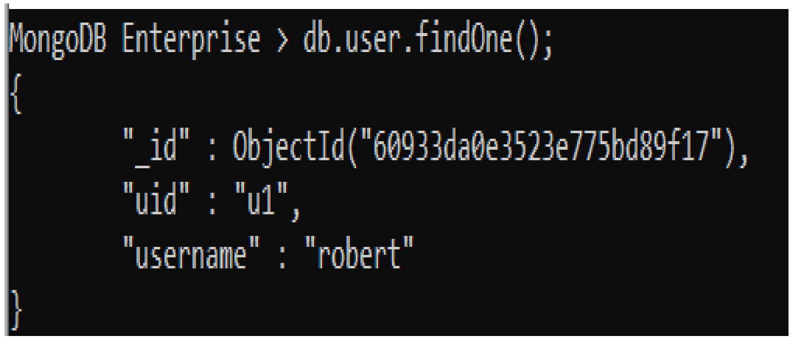
Normalized data model : In this model every sub document has an id. Normalized data models describe relationships using references between documents.

In general, use normalized data models:

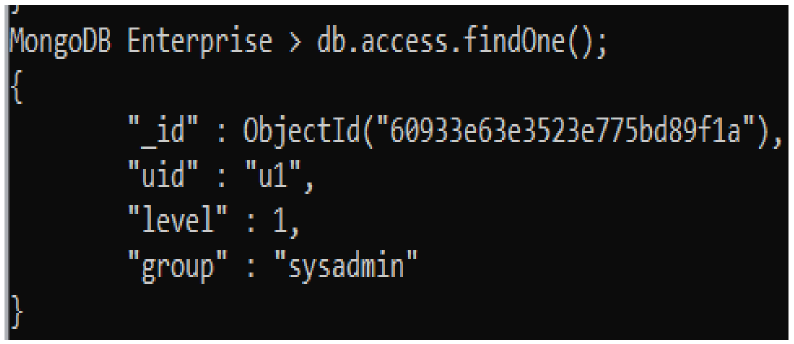
- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

Data Model in MongoDB

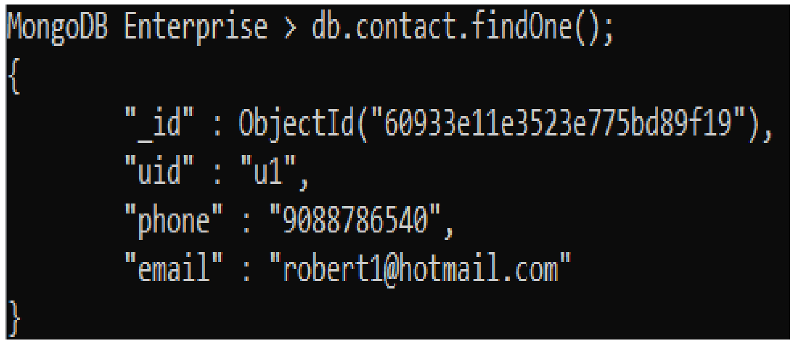
Normalized data model



```
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}
            });
```

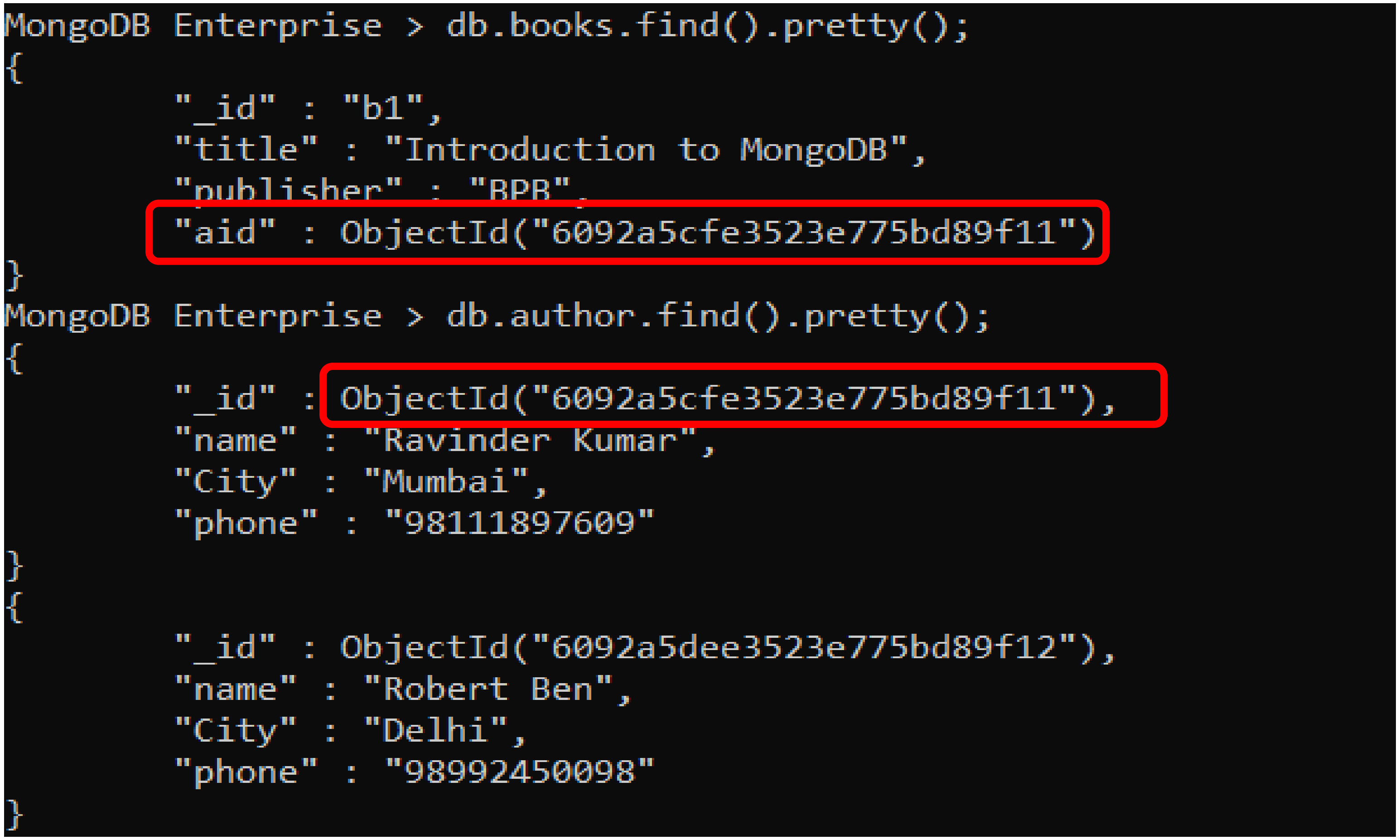
Data Model in MongoDB

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"
}
```