

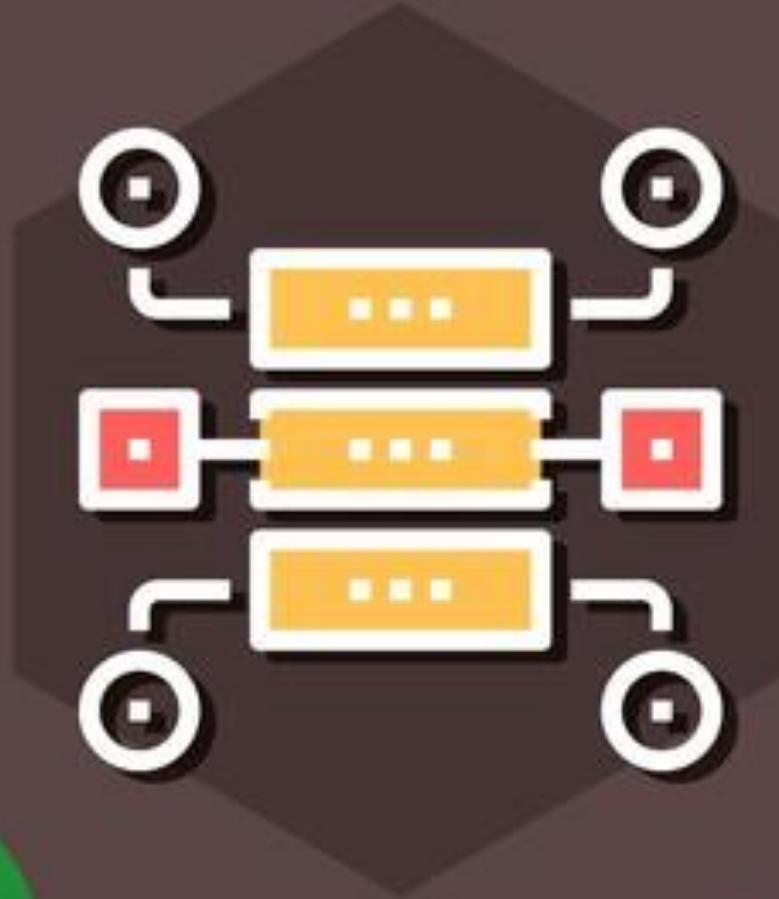


Introduction

Eman Fathi

Information Technology Institute

Data Modeling (Relationships)

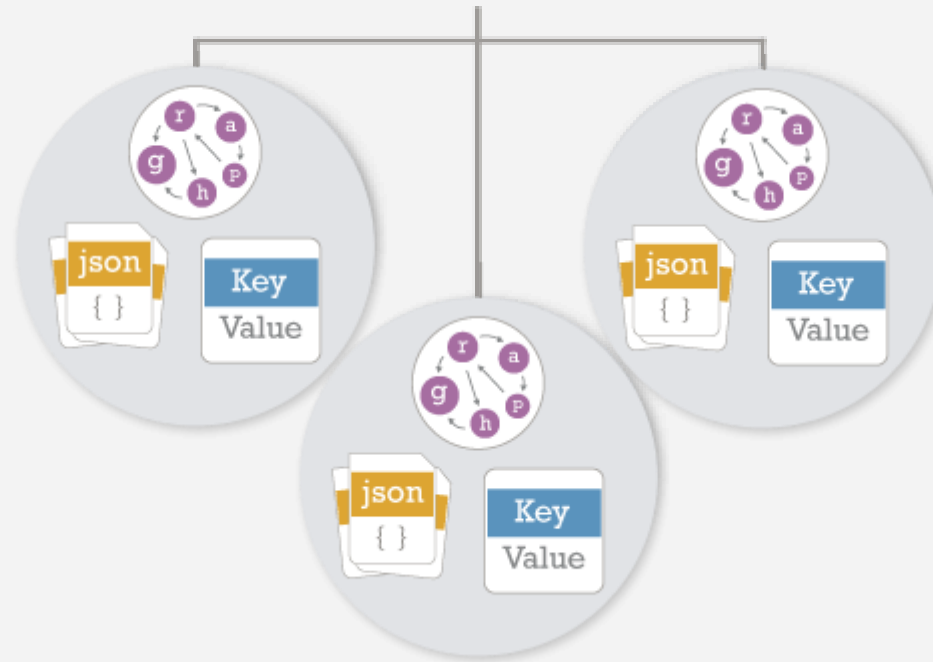


mongoDB

Data Modeling

Document Structure

- ✓ Reference
- ✓ Embedded



Consideration

- ✓ Data Usage
- ✓ Document growth

Embedding Documents

A single document can contain its own relationships.

Embedded documents capture relationships between data by storing related data in a single document structure.

MongoDB documents make it possible to embed document structures in a field or array within a document.

These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

Employee document

```
{  
  _id:1,  
  name:"Eman",  
  department: {  
    name:"SD",  
    phone:"01234"  
  }  
}
```

Embedded document



Embedding Documents



Duplicate department Information for each Employee

Employee documents

```
{
  _id:1,
  name:"Eman",
  department: {
    name:"SD",
    phone:"01234"
  }
}
```

```
{
  _id:2,
  name:"Mona",
  department: {
    name:"SD",
    phone:"01234"
  }
}
```

```
{
  _id:3,
  name:"Ali",
  department: {
    name:"SD",
    phone:"01234"
  }
}
```

```
db.employees.find({}, {dept:true, _id:false});
```

Referencing Documents

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these references to access the related data. Broadly, these are *normalized* data models.



MongoDB
does not see
anything but
value-pair
filed

Employee document

```
{
  _id : 1,
  name : "Eman",
  department_id : 3
}
```

Department document

```
{
  _id : 3,
  name : "SD",
  phone : "01234"
  email : sd@iti.com
}
```

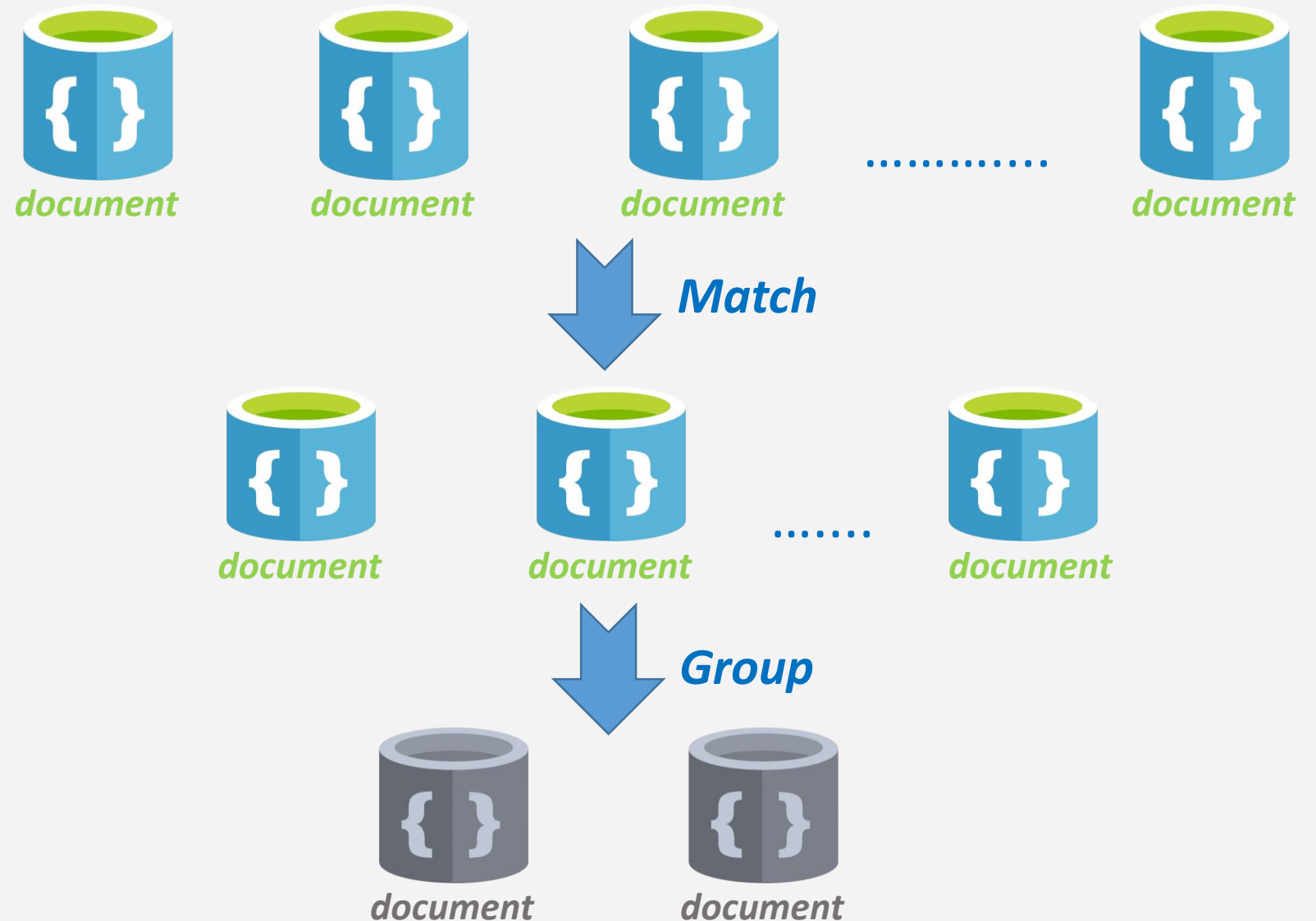
```
employee=db.employees.findOne({name:"Mohammed"});
db.departments.find({_id:employee["dept_id"]})
```

An abstract graphic featuring a dark gray background with a grid of light gray horizontal and vertical lines. Overlaid on this grid are several thick, wavy, translucent lines in various colors: purple, orange, red, and green. These lines flow vertically, with some having horizontal segments, creating a sense of movement and complexity. The lines are semi-transparent, allowing the grid lines to be seen through them.

Aggregation and
Aggregation pipelines

Aggregation

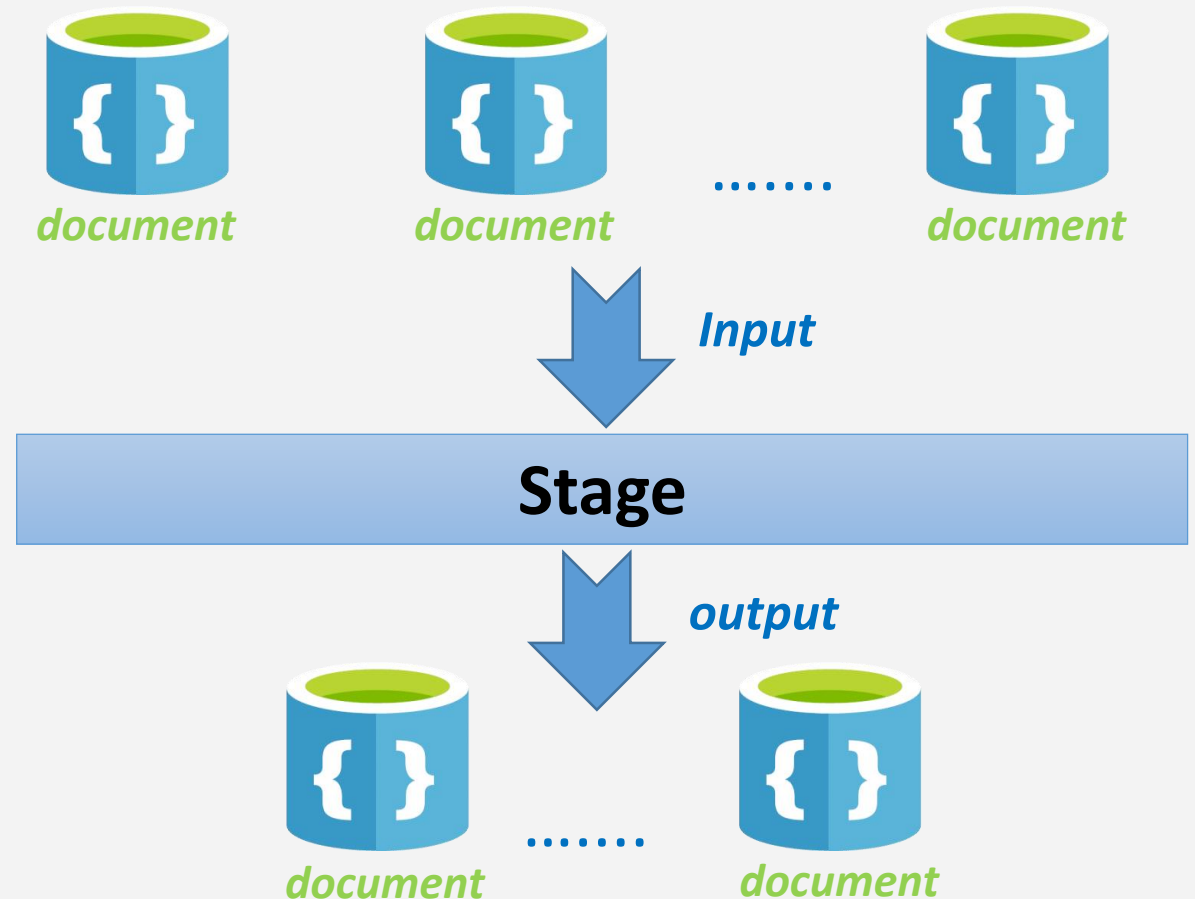
Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.



aggregate()

Documents during aggregation process pass through number of stages. Stages ([Pipelines](#)) are arrays of objects, separated by comma. Documents first are passed to stage one and the result of stage one are passed to stage two and so on..

```
db.collection.aggregate([  
  Stage1 ,  
  Stage2,  
  ...  
  ,.....  
  StageN]);
```



aggregate() Stage Operators

Each Stage starts from the stage operator `{<$StageOperator>: {} }`

```
{ $match: { age: { $gt: 20 } } }
```

```
{ $group: { _id: "$age" } }
```

```
{ $sort: { count: -1 } }
```

Name	Description
\$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s)
\$limit	Passes the first <i>n</i> documents unmodified to the pipeline where <i>n</i> is the specified limit.
\$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage.
\$out	Writes the resulting documents of the aggregation pipeline to a collection.
\$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields.
\$count	Returns a count of the number of documents at this stage of the aggregation pipeline.
\$sort	Reorders the document stream by a specified sort key

\$match

Match documents according to specified condition

`{ $match : {<query> } }`

```
db.employees.aggregate([{$match:{salary:{$gt:2000}}]})
```

```
db.employees.aggregate([{$match:{age:20}}])
```

```
db.employees.aggregate([{$match:{salary:{$gt:2000},age:20}}]);
```

\$match

Match documents according to specified condition

`{ $match : {<query> } }`

```
db.employees.aggregate([{$match:{salary:{$gt:2000}}]})
```

```
db.employees.aggregate([{$match:{age:20}}]})
```

```
db.employees.aggregate([{$match:{salary:{$gt:2000},age:20}}]);
```

Aggregation Expression

Expression refers to the name of the field in input documents **<\$FieldName>**

```
{ $group: { _id: "$age" } }
```

Result:

```
{ "_id" : 25 }  
{ "_id" : 30 }  
{ "_id" : 20 }  
{ "_id" : 40 }
```

\$group

Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an `_id` field which contains the distinct group by key. The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the `$group's` `_id` field. `$group` does *not* order its output documents.

```
{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }
```

```
db.employees.aggregate([{$group:{_id:"$age"}}]);
```

```
db.employees.aggregate([{$group:{_id:{age:"$age",salary:"$salary"}}}]);
```

```
{ $group: { _id: "$department.name" } }
```

```
{ $group: { _id: "$age", total: { $sum: "$salary" } } }
```

Accumulators

{ \$avg: <expression> }
{ \$sum: <expression> }
{ \$min: <expression> }
{ \$max: <expression> }

```
db.employees.aggregate([{$group:  
    {  
        _id:"$age",  
        salaries:{$sum:"$salary"}  
    }  
}]);
```

\$count

Passes a document to the next stage that contains a count of the number of documents input to the stage.

{ \$count:<string> }

```
db.employees.aggregate([{$count:"allDocumentsCount"}]);
```

Result: {"allDocumentsCount" : 8}

```
db.employees.aggregate([{$count:"total"}]);
```

Result: {"total" : 8}

\$sort

Sorts all input documents and returns them to the pipeline in sorted order.

```
{ $sort: { <field1>: +|-1 <field2>: +|-1... } }
```

```
db.employees.aggregate([{$sort:{name:1}}]);
```

```
db.employees.aggregate([{$sort:{name:1,age:-1}}]);
```

\$project

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

```
{ $project: {<field1>: 1|0<field2>: 1|0... } }
```

```
db.employees.aggregate([{$project:{name:1,age:1}}]);
```

```
db.employees.aggregate([{$project:{_id:0,name:1}}]);
```

```
db.employees.aggregate([{$project:{_id:0,salary:0}}]);
```

```
db.employees.aggregate([{$project:{name:1,employeeAge:"$age"}}]);
```

```
db.employees.aggregate([{$project:{name:1,
                                info:{
                                    salary:"$salary",
                                    age:"$age"
                                }
                                }}]);
```