Database Applications
Lecture 9: Data Models for MongoDB

Santha Sumanasekara
September 2022

RMIT UNIVERSITY

Topics

- Continue with queries on single document collections
- Writing queries with complex documents.
- A Data Model – **Will be discussed later**
  - Embedding
  - Referencing

RMIT UNIVERSITY

Demo time!

More queries on single document collections
Contd from week 8.

Let's do it on MongoDB Compass

Filter a document – with multiple conditions

- In SQL:

```
SELECT *
    FROM people
    WHERE status = 'A' AND
          age = 25;
```

AND is the default.

- In MongoDB Shell:

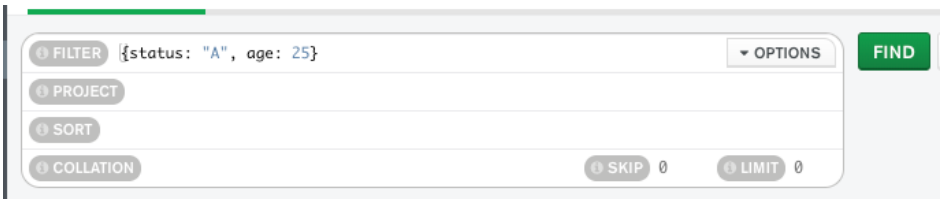
```
db.people.find(
    { status: "A",
      age: 25 }
)
```

Filter a document

- In SQL:

```
SELECT *
    FROM people
    WHERE status = 'A' AND
          age = 25;
```

- In MongoDB Compass:



RMIT UNIVERSITY

Filter a document – with OR conditions

- In SQL:

```
SELECT *
    FROM people
    WHERE status = 'A' OR
          age = 25;
```

OR is applied to an array of conditions. [ ] denotes an array.

- In MongoDB Shell:

```
db.people.find(
    { $or: [ { status: "A" } , { age: 25 } ] }
)
```

RMIT UNIVERSITY

Filter a document – with OR conditions

- In SQL:

```
SELECT *
    FROM people
    WHERE status = 'A' OR
          age = 25;
```

- In MongoDB Compass:

RMIT UNIVERSITY

Filter a document – with "Not Equal"

$ne means "not equal"

- In SQL:

```
SELECT *
    FROM people
    WHERE  age <> 25;
```

- In MongoDB Shell:

```
db.people.find(
     { age: { $ne: 25 } }
)
```

RMIT UNIVERSITY

Filter a document – middle-aged people!

- In SQL:

```
SELECT *
    FROM people
    WHERE  age > 35 AND age <= 60;
```

- In MongoDB Shell:

```
db.people.find(
     { age: { $gt: 35, $lte: 60 }}
)
```

$gt: greater than
$gte: greater than or equal
$lt: less than
$lte: less than or equal

RMIT UNIVERSITY

Filter a document – Partial Matches

- In SQL:

```
SELECT *
    FROM people
    WHERE  name LIKE '%Sam%;
```

- In MongoDB Shell:

```
db.people.find(
     {name: /Sam/}
)
```

More complex regular expressions are possible. To be discussed later. E.g.
`{name: {$regex: /Sam/ } }`

RMIT UNIVERSITY

Filter a document – Using Composite Fields

- In SQL: Cannot be done!

```
SELECT *
    FROM people
    WHERE  address.suburb = 'Happyville';
```

- In MongoDB Shell:

```
db.people.find(
     {"address.suburb": "Happyville"}
)
```

Make sure to use double quotes.

RMIT UNIVERSITY

Filter a document – Using Array Values

- In SQL: Cannot be done!

```
SELECT *
    FROM people
    WHERE  sport = ['AFL', 'Cricket']
```

- In MongoDB Shell:

```
db.people.find(
    {sports:"AFL", "Cricket"}
)
```

Return documents that exactly contain this array.

RMIT UNIVERSITY

Aggregations -- count()

- In SQL:

```
SELECT COUNT(*)
    FROM people;
```

- In MongoDB Shell:

```
db.people.find().count()
```

Object-oriented. Two ways to do it:
`db.collection.find().count()`
`db.collection.count()`

RMIT UNIVERSITY

Sorting

- In SQL:

```
SELECT *
    FROM people
    ORDER BY name;
```

- In MongoDB Shell:

```
db.people.find().sort({name: 1})
```

```
db.people.find().sort({name: -1})
```

Descending order

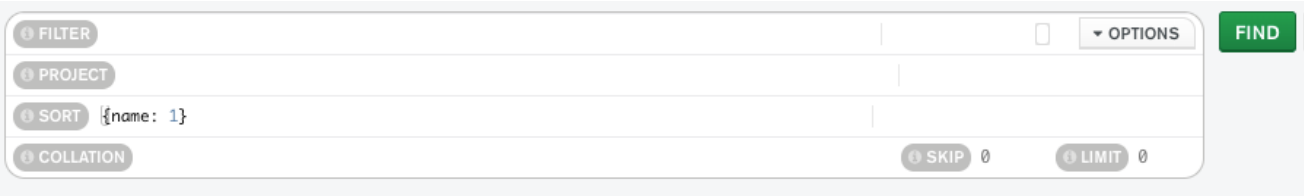
RMIT UNIVERSITY

Sorting

- In SQL:

```
SELECT *
    FROM people
    ORDER BY name;
```

- In MongoDB Compass:



RMIT UNIVERSITY

Advanced Queries using Aggregation Pipelines!

- In mongoDB, we can run queries in stages.
- Result from first stage will be the input to the second stage, and so on.
- At each stage, we carry out some aggregation operations, hence the name "Aggregation Pipeline".

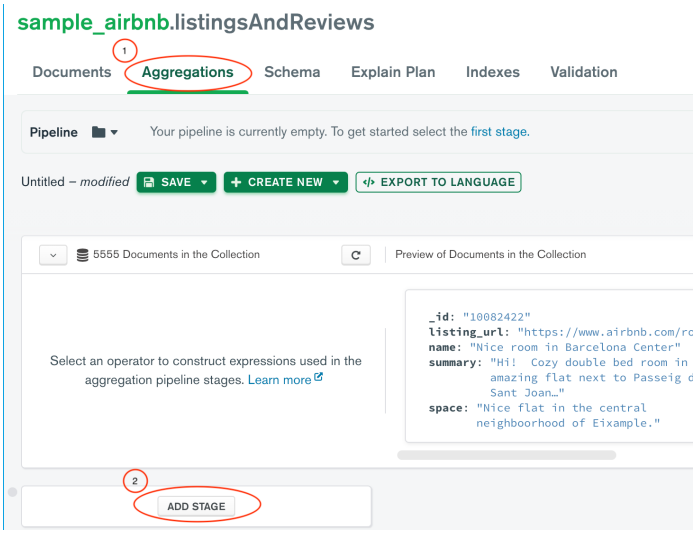
- In MongoDB Shell:

```
db.collection.aggregate(
     [ stage 1, stage 2, ...]
)
```

RMIT UNIVERSITY

Advanced Queries using Aggregation Pipelines!

- In MongoDB Compass:



Counting Documents

- It is possible to use count() operator with a simple query, which will return the number of documents in a result set.
- However, we can do more with $count: in a pipeline.

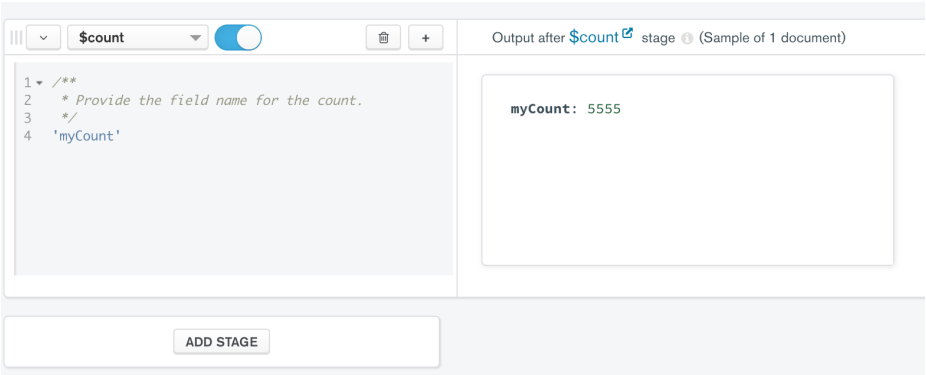
- In MongoDB Shell:

```
db.listingsAndReviews.aggregate(
                                {$count: "myCount"}
                                )
```

myCount is a new field that can be used in later stages in the pipeline.

Counting Documents

- In MongoDB Compass:



RMIT UNIVERSITY

Grouping Documents

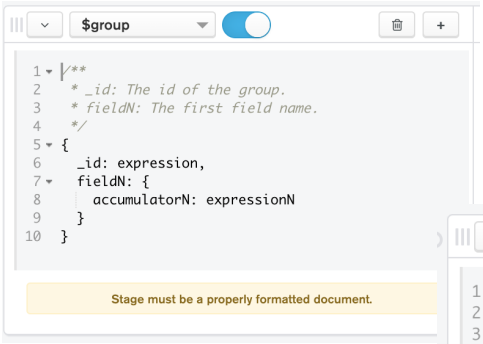
- Similar to GROUP BY clause in SQL, we can group documents based on a field (or more complex expression).
- E.g. Grouping documents on Room Type (field “room_type”)
- In MongoDB Shell:

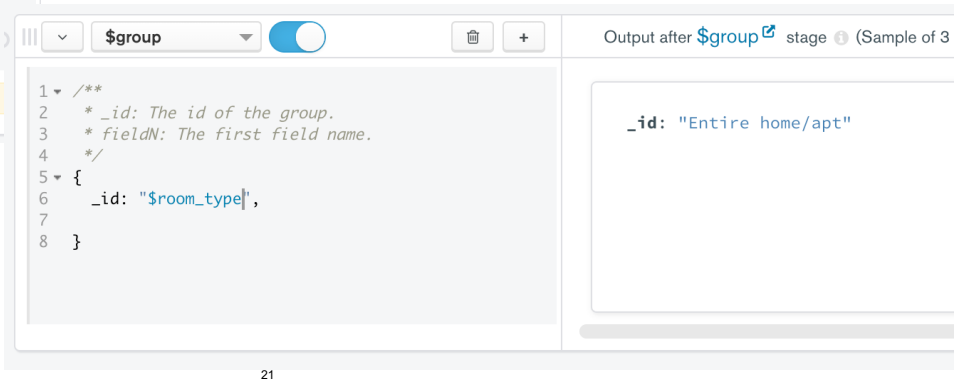
```
db.listingsAndReviews.aggregate(
                                { $group : { _id : "$room_type" } }
                                )
```

_id requires a field or expression which is used as “GROUP BY” clause.

Grouping Documents

- In MongoDB Compass:





RMIT UNIVERSITY

Grouped Aggregation

- As shown in the previous example, grouping will only show one document each showing the field used for grouping. – not much useful.
- Grouped aggregation uses both grouping and an aggregation, such as counting, together.
- E.g. Count no. of listings for each room type

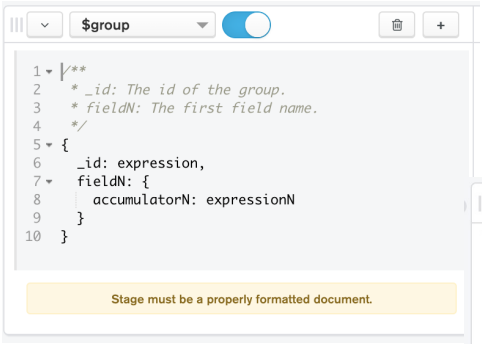
- In MongoDB Shell:

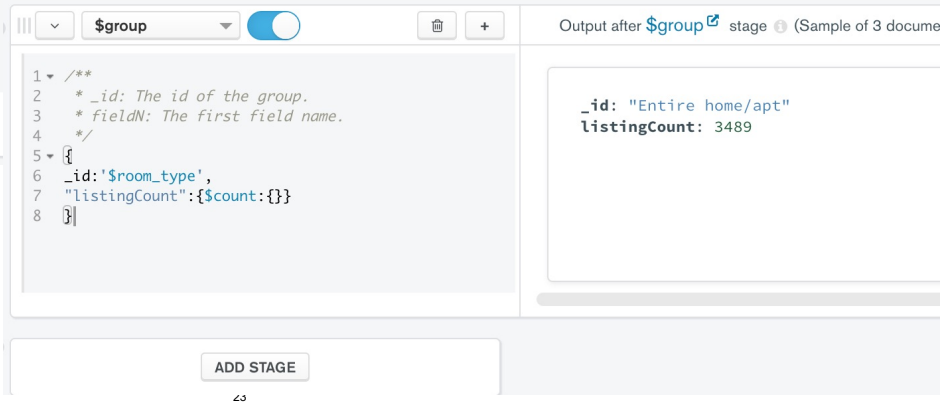
```
db.listingsAndReviews.aggregate(
                                  { $group : { _id : "$room_type",
                                    "listingCount":{$count:{}} } }
                                )
```

RMIT UNIVERSITY

Grouped Aggregation

- In MongoDB Compass:





Filtering on Grouped Aggregation

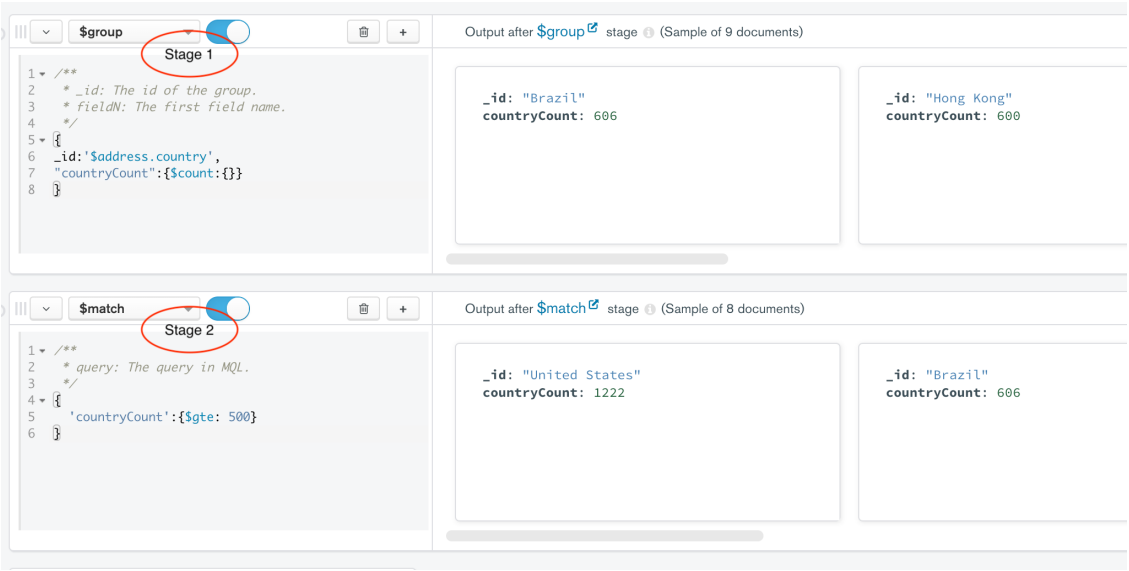
- Similar to HAVING clause in SQL, we can filter on the result of a group aggregation.
- E.g. Show the countries with more than 500 listings.
- First, group on country, then count, and finally filter on the count.

- In MongoDB Shell:

```
db.listingsAndReviews.aggregate( [ { $group :
                                      { _id : "$address.country",
                                       "countryCount":{$count:{}} }
                                      },
                                      { $match: { "countryCount":
                                                    { $gte: 500 } }
                                  }
                                  ] )
```

Grouped Aggregation

- In MongoDB Compass:



More complex groupings

- MongoDB (with pipelines) allows us to make complex groupings possible
- E.g. Count listings in $100 price brackets.
  - Stage 1: build a suitable expression to be used for grouping
  - Stage 2: Use the expression generated in Stage 1 for grouping

```
db.listingsAndReviews.aggregate( [ {
$project:
{
  "price":1,
  priceRange: { $round:{$divide: [ "$price", 100 ] }}
}},
{
$group:
{
_id:'$priceRange',
"RangeCount":{$count:{}}
}
}])
```

More complex groupings

$project

```
/**
 * specifications: The fields to
 *   include or exclude.
 */
{
  "price":1,
  priceRange: { $round:{$divide: [ "$price", 100 ] }
}
```

Output after $project stage (Sample of 10 documents)

```
_id: "10082422"
price: 50.00
priceRange: 0
```

```
_id: "1003530"
price: 135.00
priceRange: 1
```

$group

```
/**
 * _id: The id of the group.
 * fieldN: The first field name.
 */
{
_id:'$priceRange',
"RangeCount":{$count:{}}
}
```

Output after $group stage (Sample of 10 documents)

```
_id: 11
RangeCount: 24
```

```
_id: 10
RangeCount: 52
```

$sort

```
/**
 * Provide any number of field/order pairs.
 */
{
  _id: 1
}
```

Output after $sort stage (Sample of 10 documents)

```
_id: 0
RangeCount: 868
```

```
_id: 1
RangeCount: 2235
```

More complex groupings

- Amenities is an array. If you wish to group documents by the amenities offered, you will have to dis-integrate the array into individual values and then apply the $group on the result.
- Use $unwind: to disintegrate an array.

```
db.listingsAndReviews.aggregate( [ {$unwind: {
                                        path: "$amenities" }
                                   },
                                   {$group :
                                     { _id : "$amenities",
                                     "amenitiesCount":{$count:{}} }
                                   }
                                    ] )
```

More complex groupings

$unwind

```
/**
 * path: Path to the array field.
 * includeArrayIndex: Optional name for index.
 * preserveNullAndEmptyArrays: Optional
 *   toggle to unwind null and empty values.
 */
{
  path: "$amenities"
}
```

Output after $unwind stage (Sample of 10 documents)

```
_id: "10082422"
listing_url: "https://www.airbnb.com/rooms/1
name: "Nice room in Barcelona Center"
summary: "Hi!  Cozy double bed room in
          amazing flat next to Passeig de
          Sant Joan…"
space: "Nice flat in the central
        neighboorhood of Eixample."
```

```
_id: "10082422"
listing_url: "https://www.air
name: "Nice room in Barcelona
summary: "Hi!  Cozy double be
          amazing flat next to
          Sant Joan…"
space: "Nice flat in the cent
        neighboorhood of Eixam
```

$group

```
/**
 * _id: The id of the group.
 * fieldN: The first field name.
 */
{
_id:'$amenities',
"amenitiesCount":{$count:{}}
}
```

Output after $group stage (Sample of 10 documents)

```
_id: "Coffee maker"
amenitiesCount: 1450
```

```
_id: "Free parking on premise
amenitiesCount: 1489
```

$sort

```
/**
 * Provide any number of field/order pairs.
 */
{
  amenitiesCount: -1
}
```

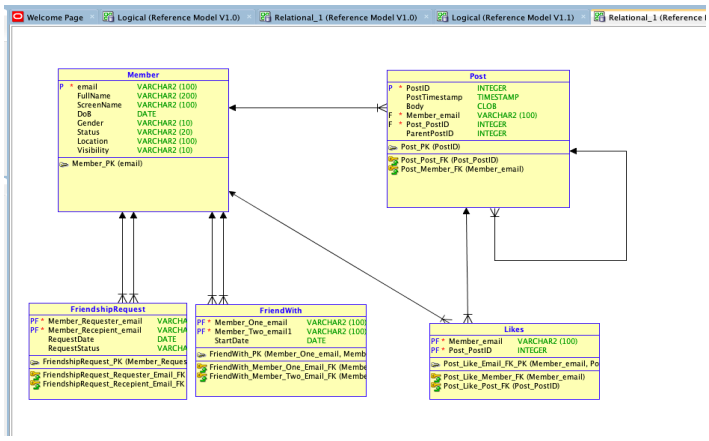
Output after $sort stage (Sample of 10 documents)

```
_id: "Wifi"
amenitiesCount: 5303
```

```
_id: "Essentials"
amenitiesCount: 5048
```

What is a data model?

- It is clear in the relational model.
- The database schema defines the underlying data model.



RMIT UNIVERSITY

What is a data model?

- In non-relational context, the data model is not explicit.
- Unlike in the relational model, it is not a requirement to define the data model up-front.
- MongoDB's collections, by default, does not require its documents to have the same schema.
  - However, in practice, the documents in a collection share a similar structure.

Recall AirBnB database. Not all documents have the same fields, but, many of them are common!

RMIT UNIVERSITY

Document Structure

- The data model, in a non-relational context, is based on the structure of the documents:
  - "Document" is considered as an atomic building block of the data model.
- When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

RMIT UNIVERSITY

Document Structure

- Simple documents
  - Captures and stores facts about a single entity.
  - In MongoDB, such documents are collated and stored in one document collection.

We are familiar with such documents. We created, populated and queried such documents last week.

RMIT UNIVERSITY

Document Structure

- Simple documents

MyDB.Movies

Documents | Aggregations | Explain Plan | Indexes

FILTER

INSERT DOCUMENT | VIEW | LIST | TABLE

Movies

	_id ObjectId	mvNumb Int32	mvTitle String	yrMade Int32
1	5d8b77a782da1eb3add62925	1	"Annie Hall"	1977
2	5d8b784482da1eb3add62926	2	"Dr. Strangelove"	1964
3	5d8b78bb82da1eb3add62927	7	"Interiors"	1978

Document Structure

- Complex documents
  - Most of the time, our applications require storing information about many “inter-related” entities.
  - Consider information about movies and directors.
  - MongoDB data model handles such complex documents in two ways:
    - Embedding sub-documents
    - Referencing from one document to another.

RMIT UNIVERSITY

Document Structure -- Embedded

- MongoDB allows related data to be embedded within a single document.
- MongoDB allows us to embed sub-documents in a field or array within a document.
- These denormalized data models allow applications to retrieve and manipulate related data way faster than the relational model, as there aren’t any JOIN operations.

RMIT UNIVERSITY

An Example

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

RMIT UNIVERSITY

Another Example

```
_id: ObjectId("5d8b8a5882da1eb3add6292b")
username: "dba"
email: "dba@example.com"
address: Array
  0: Object
     streetNo: "1"
     streetName: "Happy Street"
     suburb: "Happyville"
     postCode: "3999"
     state: "VIC"
  1: Object
     streetNo: "2"
     streetName: "Jollymont Ave"
     suburb: "Tigerland"
     postCode: "3990"
     state: "VIC"
```

This particular document has two sub-documents embedded within the address field. Each sub-document has its own document structure.

Document Structure -- Embedded

- In general, use embedded data models when:
  - you have “contains” (or “has”) relationships between entities.
  - you have one-to-many relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents. E.g. directors → movies

```
_id: ObjectId("5d8b793882da1eb3add62928")
dirNumb: 1
dirName: "Allen, Woody"
dirBorn: 1935
director_movies: Array
  0: Object
  1: Object
```

RMIT UNIVERSITY

Another Example

Will discuss about this example later!

```
_id: ObjectId("5d8b77a782da1eb3add62925")
mvNumb: 1
mvTitle: "Annie Hall"
movie_directors: Array
  0: Object

    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

RMIT UNIVERSITY

Querying with Embedded Documents

- **Querying is easy with embedded sub-documents**.
- Each sub-document is treated as an object or an array.
- Use [ ] (square bracket) or . (dot) to refer them in filtering, projections, etc.

```
db.movies.find({"Director.dirName": "Allen, Woody"})
```

or

```
db.movies.find({"Star.starName": "Allen, Woody"})
```

RMIT UNIVERSITY

Drawbacks of Embedding

- Embedding a sub-document is like "de-normalising" a document.
- This can lead into redundancies, and that can result in undesirable anomalies (insertion, update, delete, etc).
- While you gain performance gains (by avoiding JOINs), embedding is not suited for some instances.
- An example is when documents are in a many-to-many relationships, or, in a one-to-many relationship where most of the queries are based on many side.

RMIT UNIVERSITY

Drawbacks of Embedding

```
_id: ObjectId("5d8b78bb82da1eb3add62927")
mvNumb: 7
mvTitle: "Interiors"
movie_directors: Array
  0: Object

    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

```
_id: ObjectId("5d8b77a782da1eb3add62925")
mvNumb: 1
mvTitle: "Annie Hall"
movie_directors: Array
  0: Object

    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

Redundant data

RMIT UNIVERSITY

Drawbacks of Embedding

```
_id: ObjectId("5d8b78bb82da1eb3add62927")
mvNumb: 7
mvTitle: "Interiors"
movie_directors: Array
  0: Object

    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

```
_id: ObjectId("5d8b77a782da1eb3add62925")
mvNumb: 1
mvTitle: "Annie Hall"
movie_directors: Array
  0: Object

    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

If the performance gains outweigh the troubles of managing the redundancy anomalies, MongoDB prefers this (denormalised) data model!

RMIT UNIVERSITY

Drawbacks of Embedding

```
_id: ObjectId("5d8b78bb82da1eb3add62927")
mvNumb: 7
mvTitle: "Interiors"
movie_directors: Array
  0: Object
    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

```
_id: ObjectId("5d8b77a782da1eb3add62925")
mvNumb: 1
mvTitle: "Annie Hall"
movie_directors: Array
  0: Object
    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

In IMDB application, many more queries are executed based on movies, compared to queries on directors. So, the above can still be the desirable data model.

Drawbacks of Embedding

```
_id: ObjectId("5d8b78bb82da1eb3add62927")
mvNumb: 7
mvTitle: "Interiors"
movie_directors: Array
  0: Object
    _id: ObjectId("5d8b793882da1eb3add62928")
    dirNumb: 1
    dirName: "Alle  ody"
    dirBorn: 1935
```

```
_id: ObjectId("5d8b77a782da1eb3add629
mvNumb: 1
mvTitle: "Annie Hall"
movie_directors: Array
  0: Object
    _id: ObjectId("5d8b793882da1eb3ac
    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

If redundancy poses a bigger problem, consider “referencing”!

Document Structure -- Referencing

- This is a ”normalised” data model, where relationships between entities are established between documents using references.

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

RMIT UNIVERSITY

Another Example

MyDB.Movies

Documents | Aggregations | Explain Plan

FILTER
PROJECT
SORT {yrMade: -1}
COLLATION

INSERT DOCUMENT VIEW LIST TABLE

```
_id: ObjectId("5d8b78bb82da1eb3add62927")
mvNumb: 7
mvTitle: "Interiors"
yrMade: 1978
mvType: "DRAMA"
dirNumb: 1
```

```
_id: ObjectId("5d8b77a782da1eb3add62925")
mvNumb: 1
mvTitle: "Annie Hall"
yrMade: 1977
mvType: "COMEDY"
crit: 4
MPAA: 5
noms: 5
awrd: 4
dirNumb: 1
```

MyDB.Directors

Documents | Aggregations | Explain Plan

FILTER

INSERT DOCUMENT VIEW LIST TABLE

```
_id: ObjectId("5d8b793882da1eb3add62928")
dirNumb: 1
dirName: "Allen, Woody"
dirBorn: 1935
```

```
_id: ObjectId("5d8b795e82da1eb3add62929")
dirNumb: 2
dirName: "Hitchcock, Alfred"
dirBorn: 1899
dirDied: 1980
```

```
_id: ObjectId("5d8b799882da1eb3add6292a")
dirNumb: 5
dirName: "Kubrick, Stanley"
dirBorn: 1928
```

Another Example

```
_id: ObjectId("5d8b78bb82da1eb3add62927")
mvNumb: 7
mvTitle: "Interiors"
movie_directors: Array
  0: Object
    _id: ObjectId("5d8b793882da1eb3add62928")
    dirNumb: 1
    dirName: "Alle  ody"
    dirBorn: 1935
```

```
_id: ObjectId("5d8b77a782da1eb3add629
mvNumb: 1
mvTitle: "Annie Hall"
movie_directors: Array
  0: Object
    _id: ObjectId("5d8b793882da1eb3ac
    dirNumb: 1
    dirName: "Allen, Woody"
    dirBorn: 1935
```

Information about Woody Allen are stored in a document with _id "5d8b79388..." and all his movies refers to this document. Woody's information stored once.

Document Structure -- Referencing

- In general, use normalised 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.

How about "posts" in our Facebook-Lite application?

Querying with Referenced Documents

- **Querying is not straight-forward with referenced documents**.
- Early versions of MongoDB didn't even support referencing.
- Starting from MongoDB 3.2 (current version is 4.2), **$lookup()** function allows us to write queries to retrieve information from referenced documents.

RMIT UNIVERSITY

Querying with $lookup()

- Querying with $lookup() can be a part of "Aggregation Pipeline"
- Starts from one-side of the join, then state what document collection to be in the many-side and local and destination fields to match.

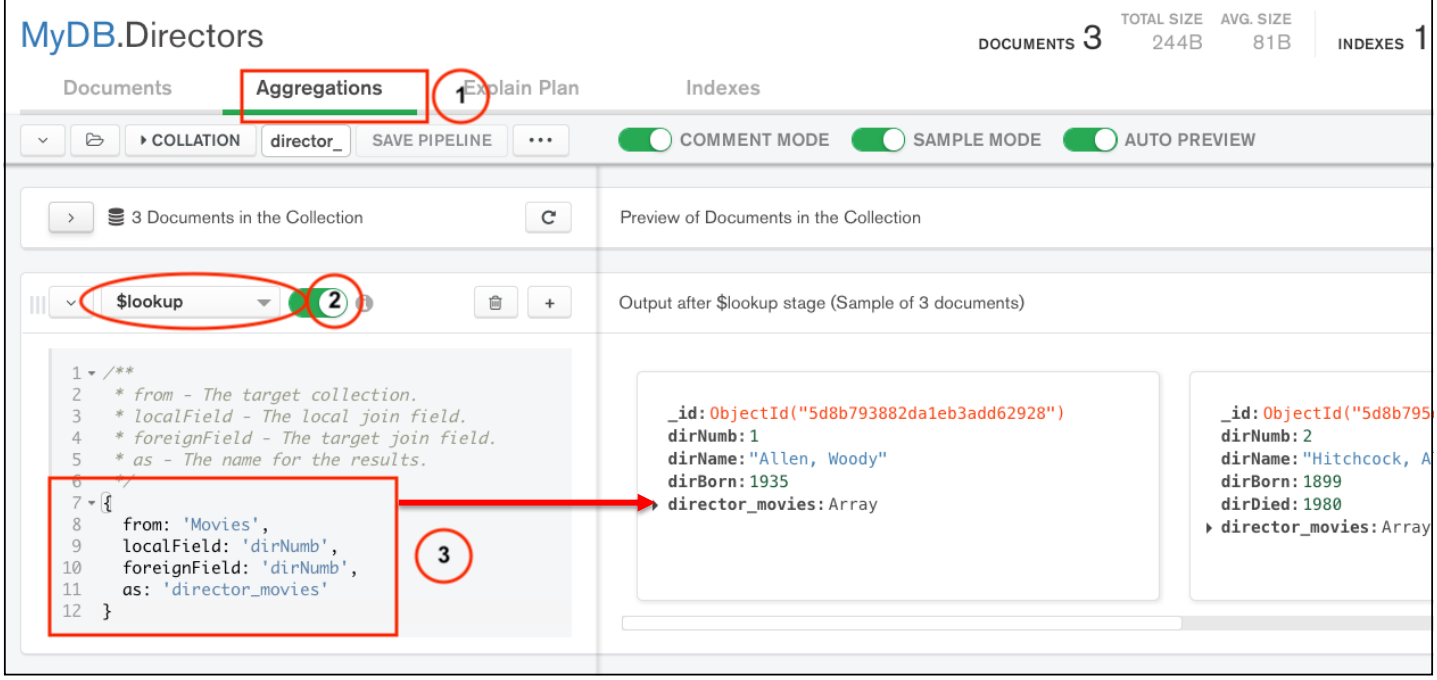
```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Querying with $lookup() – An example

- Let's suppose directors have many movies.
- So, join starts from "Directors" collection, destination collection is "Movies".
- Common field is "dirNumb" (on both collections, but can have different fields)

```
{
   $lookup:
     {
       from: Movies,
       localField: dirNumb,
       foreignField: dirNumb,
       as: director_movies
     }
}
```


Querying with $lookup() – An example



Demo time!

Referencing

Let's do it on MongoDB Compass

RMIT UNIVERSITY

Querying with $lookup()

- For more details and complex operations, see:

  https://docs.mongodb.com/manual/reference/operator/aggregation/lookup

RMIT UNIVERSITY

Further Readings

- https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/
- https://docs.mongodb.com/php-library/current/tutorial/
- https://docs.mongodb.com/ecosystem/drivers/php/

RMIT UNIVERSITY

Next Week

- We continue this MongoDB discussion in Week 10. We discuss on replication, etc.

RMIT UNIVERSITY

Menti-time!

Menti.com
Code: to be supplied

RMIT UNIVERSITY

Questions?

RMIT UNIVERSITY