Big Data (MHI222956/MHI225101)

4.1 Advanced querying in MongoDB

An example document in the *restaurants* collection


```
{
        "_id" : ObjectId("59e565aa8b8edcdd5b7a43bc"),
        "address" : {
                "building" : "469",
                "coord" : [
                        -73.961704,
                        40.662942
                ],
                "street" : "Flatbush Avenue",
                "zipcode" : "11225"
        },
        "borough" : "Brooklyn",
        "cuisine" : "Hamburgers",
        "grades" : [
                {
                        "date" : ISODate("2014-12-30T00:00:00Z"),
                        "grade" : "A",
                        "score" : 8
                },
                {
                        "date" : ISODate("2014-07-01T00:00:00Z"),
                        "grade" : "B",
                        "score" : 23
                },
                {
                        "date" : ISODate("2013-04-30T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                },
                {
                        "date" : ISODate("2012-05-08T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                }
        ],
        "name" : "Wendy'S",
        "restaurant_id" : "30112340"
}
```

```
{
        "_id" : ObjectId("59ee697aaa1d2db3dc4fbfa5"),
        "title" : "Star Wars: Episode II - Attack of the Clones",
        "year" : 2002,
        "rated" : "PG",
        "runtime" : 142,
        "countries" : [
                "USA"
        ],
        "genres" : [
                "Action",
                "Adventure",
                "Fantasy"
        ],
        "director" : "George Lucas",
        "writers" : [
                "George Lucas",
                "Jonathan Hales",
                "George Lucas"
        ],
        "actors" : [
                "Ewan McGregor",
                "Natalie Portman",
                "Hayden Christensen",
                "Christopher Lee"
        ],
        "plot" : "Ten years after initially meeting, Anakin Skywalker shares a f
orbidden romance with Padmé, while Obi-Wan investigates an assassination attempt
 on the Senator and discovers a secret clone army crafted for the Jedi.",
        "poster" : "http://ia.media-imdb.com/images/M/MV5BMTY5MjI5NTIwNl5BMl5Ban
BnXkFtZTYwMTM1Njg2._V1_SX300.jpg",
        "imdb" : {
                "id" : "tt0121765",
                "rating" : 6.7,
                "votes" : 425728
        },
        "tomato" : {
                "meter" : 66,
                "image" : "fresh",
                "rating" : 6.7,
                "reviews" : 242,
                "fresh" : 159,
                "consensus" : "Star Wars Episode II: Attack of the Clones benefi
ts from an increased emphasis on thrilling action, although they're once again u
ndercut by ponderous plot points and underdeveloped characters.",
                "userMeter" : 58,
                "userRating" : 3.3,
                "userReviews" : 844634
        },
        "metacritic" : 54,
        "awards" : {
                "wins" : 13,
                "nominations" : 47,
                "text" : "Nominated for 1 Oscar. Another 13 wins & 47 nomination
s."
        },
        "type" : "movie"
}
```


An example document in the movieDetails collection

Query on Embedded Documents

- When your data model has embedded documents you often need to query on the values in fields of embedded objects
- Use dot notation ("*field.nestedField*") to specify a query condition on fields in an embedded document:

```
> db.restaurants.find({"address.street":"Flatbush Avenue"},{"name":1,"_id":0}).l
imit(10)
{ "name" : "Wendy'S" }
{ "name" : "New Floridian Diner" }
{ "name" : "Mcdonald'S" }
{ "name" : "Geido Restaurant" }
{ "name" : "Frank'S Pizza Restaurant" }
{ "name" : "Antonio'S Pizza" }
{ "name" : "La Cabana Restaurant" }
{ "name" : "Gino'S Pizza" }
{ "name" : "Crystal Manor" }
{ "name" : "Family Pizza" }
>
```

- Note that you can often omit quotes round field names with no error, but embedded field names using dot notation need quotes and you get an error if omitted

Query Arrays

- To specify equality condition on an array, use the query document { <field>: <value> } where <value> is the exact array to match, including the order of the elements.

```
>db.movieDetails.find({countries:["Italy","USA"]}, {countries:1})
```

```
> db.movieDetails.find({countries:["Italy","USA"]}, {countries:1})
{ "_id" : ObjectId("59e920f28eeffc9227fa41ce"), "countries" : [ "Italy", "USA" ]
}
{ "_id" : ObjectId("59e920f28eeffc9227fa4297"), "countries" : [ "Italy", "USA" ]
}
{ "_id" : ObjectId("59e920f28eeffc9227fa4625"), "countries" : [ "Italy", "USA" ]
}
{ "_id" : ObjectId("59e920f28eeffc9227fa49e9"), "countries" : [ "Italy", "USA" ]
}
>
```

Query Arrays

- $all operator, this gives documents with all the specified values, without regard to order or other elements in the array.

```
>db.movieDetails.find({countries: {$all:["Italy","USA"]}}, {countries:1})
```

```
> db.movieDetails.find({countries: {$all:["Italy","USA"]}}, {countries:1})
{ "_id" : ObjectId("59e920f28eeffc9227fa418b"), "countries" : [ "Italy", "USA",
"Spain" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa41ce"), "countries" : [ "Italy", "USA" ]
 }
{ "_id" : ObjectId("59e920f28eeffc9227fa424e"), "countries" : [ "USA", "France",
 "Italy", "Germany" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4297"), "countries" : [ "Italy", "USA" ]
 }
{ "_id" : ObjectId("59e920f28eeffc9227fa4625"), "countries" : [ "Italy", "USA" ]
 }
{ "_id" : ObjectId("59e920f28eeffc9227fa49c1"), "countries" : [ "USA", "Luxembou
rg", "France", "Italy" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa49e9"), "countries" : [ "Italy", "USA" ]
 }
>
```

Query Arrays

- *$in* operator, this gives documents with any of the specified values in the authors array (note that *$in* operator can also be used with a single value field)

```
>db.movieDetails.find({countries: {$in:["Italy","USA"]}},
{countries:1}).limit(10)
```

```
> db.movieDetails.find({countries: {$in:["Italy","USA"]}}, {countries:1}).limit(
10)
{ "_id" : ObjectId("59e920f28eeffc9227fa418b"), "countries" : [ "Italy", "USA",
"Spain" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa418c"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa418d"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa418e"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4190"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4191"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4192"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4194"), "countries" : [ "New Zealand", "
USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4195"), "countries" : [ "USA" ] }
{ "_id" : ObjectId("59e920f28eeffc9227fa4196"), "countries" : [ "USA" ] }
>
```

Query Complex Objects

- Can query documents with complex structures – need to take care to get exactly the data you want
- This query finds all restaurants where the *grades* array contains an object whose *score* field contains the value 7, and returns only the *name* field and the *score* field of all the objects in the *grades* array (Note *findOne* instead of *find*, returns one document only)

```
> db.restaurants.findOne({"grades.score":7},{"name":1,"grades.score":1,"_id":0})

{
        "grades" : [
                {
                        "score" : 5
                },
                {
                        "score" : 7
                },
                {
                        "score" : 12
                },
                {
                        "score" : 12
                }
        ],
        "name" : "Riviera Caterer"
}
>
```

Querying with Regular Expressions

- Regular expression using $regex providing capabilities for pattern matching strings in queries.
- Similar to *wildcards* in SQL; for further details:

  https://docs.mongodb.com/manual/reference/operator/query/regex/

e.g. Find the names of 5 restaurants starting with 'cafe'

```
>db.restaurants.find({"name": {$regex: /^cafe/i }},{"name":1, "_id":0}).limit(5)
```

Prefix expression

Case insensitive

```
> db.restaurants.find({"name": {$regex: /^cafe/i }},{"name":1, "_id":0}).limit(5
)
{ "name" : "Cafe Metro" }
{ "name" : "Cafe Atelier (Art Students League)" }
{ "name" : "Cafe Riazor" }
{ "name" : "Cafe Un Deux Trois" }
{ "name" : "Cafe Capri" }
>
```

Aggregation

- 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:
  - Single purpose aggregation methods
  - Aggregation pipeline
  - Map-reduce function

Single purpose aggregation methods

- db.collection.count()
  - Counts the number of documents in a collection or a view.
  - e.g., finds the number of movies with year 2009

```
> db.movieDetails.find({year:2009}).count()
2
```

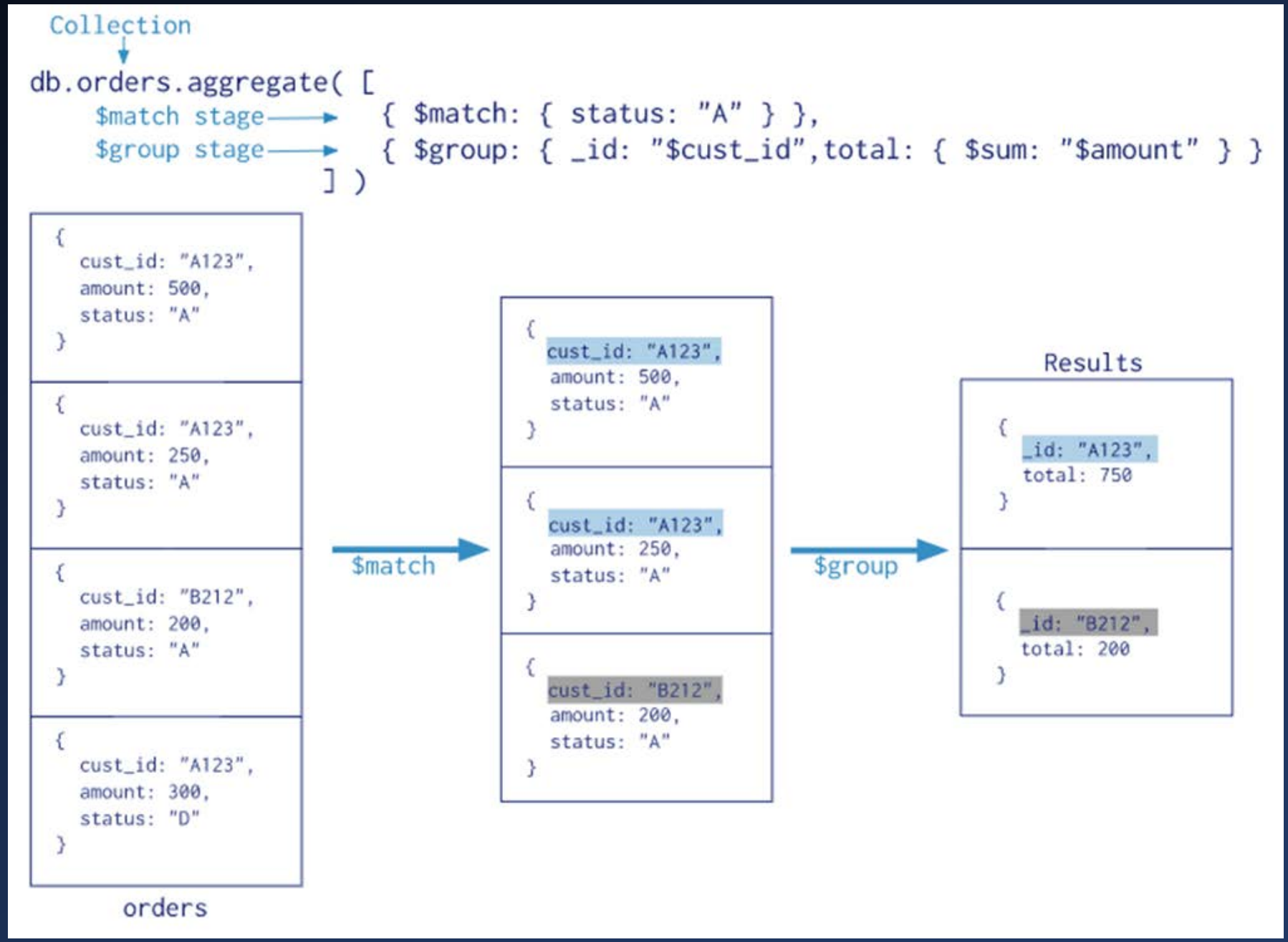
- db.collection. distinct ()
  - Displays the distinct values found for a specified key in a collection or a view.
  - e.g., finds the number of distinct director names

```
> db.movieDetails.distinct("director").length
1870
```

Aggregation Pipeline

- A framework for data aggregation modelled on the concept of data processing pipelines.
- The MongoDB aggregation pipeline consists of stages. Each stage transforms the documents as they pass through the pipeline. Documents enter the *multi-stage* pipeline that transforms the documents into *aggregated results*.
- Pipeline stages defined with *operators*, such as *$match*, *$group*, and appear in an array.
- Pipeline stages do not need to produce one output document for every input document; e.g., some stages may generate new documents or filter out documents.
- Pipeline stages can appear multiple times in the pipeline.

- An ggregation Pipeline example



- Aggregate pipeline method:

  >db.collection.aggregate([{ <stage> }, ...])

- Pipeline Operators:

Name	Description
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.

- When using $group, each group must have an _id field
- If you want to apply aggregation to all the documents use null for _id

- Further details on Aggregation Pipeline:

  https://docs.mongodb.com/manual/core/aggregation-pipeline/

Aggregation Pipeline Examples

- Example with ZIP code Data


```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```


- Aggregation operation returns all states with total population greater than 10 million:

```
db.zipcodes.aggregate( [
    { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
    { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

```
{ "_id" : "TX", "totalPop" : 16986510 }
{ "_id" : "IL", "totalPop" : 11430602 }
{ "_id" : "CA", "totalPop" : 29760021 }
{ "_id" : "OH", "totalPop" : 10847115 }
{ "_id" : "FL", "totalPop" : 12937926 }
{ "_id" : "NY", "totalPop" : 17990455 }
{ "_id" : "PA", "totalPop" : 11881643 }
```

- Aggregation operation returns total population of the USA

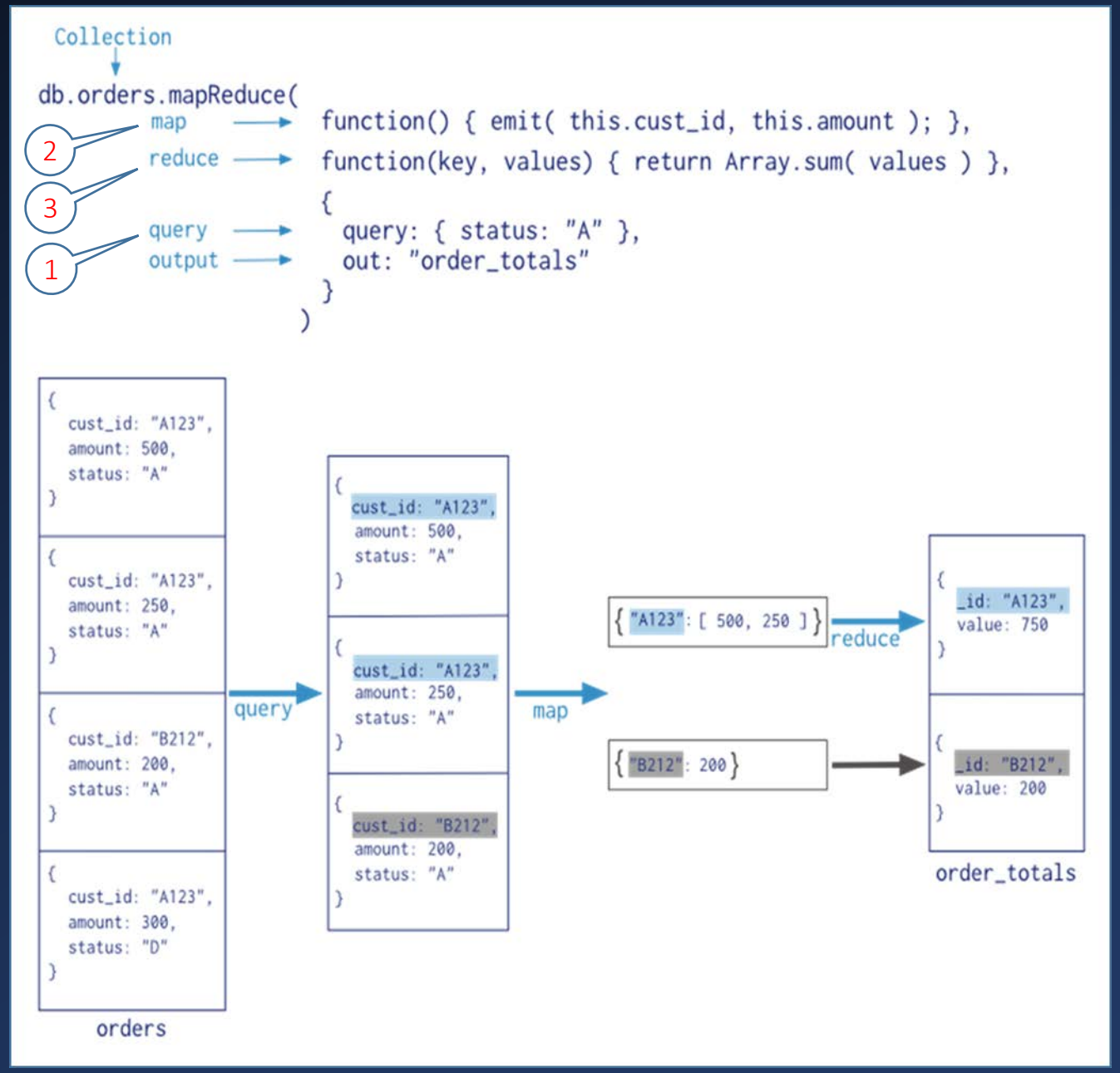
```
db.zips.aggregate([
...     { $group: {_id: null, totalPop: { $sum:
"$pop" }}}
... ])
```

```
{ "_id" : null, "totalPop" : 248706415 }
```

Map-Reduce

- A data processing paradigm for condensing large volumes of data into useful *aggregated* results.
- For map-reduce operations, MongoDB provides the *mapReduce* database command. Map-reduce operations take the documents of a single collection as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage.
- MapReduce can return the results of a map-reduce operation as a document, or may write the results to collections.
- All map-reduce functions are written in *JavaScript* and run within the mongod process.

- This example perform the same aggregation as the previous diagram but using map-reduce
- Query runs first to select documents to map (equivalent to *$match*)
- Map function emits each key with the values for the amount field
- Reduce function sums the amounts, passed in as an array, for each key
- Map and reduce together equivalent to *$group*



- Map function
  - First queries the collections then maps the result documents to emit key-value pairs

```
> var mapFunction = function() {
... emit(this.state, this.pop);
... };
```

- Reduce function
  - These are reduced based on the keys that have multiple values

```
> var reduceFunction = function(keyId,
      valuesPops) {
... return Array.sum(valuesPops);
... };
```

define map and reduce functions – map emits state as key, population as value for each document, one pair emitted for each document in data

```
> db.zips.mapReduce(
... mapFunction,
... reduceFunction,
... {
... out: "map_reduce_example"
... }
... )
```

do map-reduce, output to a collection *map_reduce_exampl*

get information on map-reduce job – in this case 29467 documents were emitted (number of zip codes) and 51 output after reduce (number of states)

```
{
        "result" : "map_reduce_example",
        "timeMillis" : 239,
        "counts" : {
                "input" : 29467,
                "emit" : 29467,
                "reduce" : 348,
                "output" : 51
        },
        "ok" : 1
}
```

- Can use *find()* function on output collection to see results
- In this case define a further query to get all states with total population greater than 10 million (same as aggregation pipeline example earlier)

```
> db.zips.mapReduce(
...   mapFunction,
...   reduceFunction,
...   {
...   out: "map_reduce_example"
... }
... ).find({value: {$gt: 10000000}})

{ "_id" : "CA", "value" : 29760021 }
{ "_id" : "FL", "value" : 12937926 }
{ "_id" : "IL", "value" : 11430602 }
{ "_id" : "NY", "value" : 17990455 }
{ "_id" : "OH", "value" : 10847115 }
{ "_id" : "PA", "value" : 11881643 }
{ "_id" : "TX", "value" : 16986510 }
```

Map-reduce vs Aggregation Pipeline

- For most aggregation operations, the Aggregation Pipeline provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.
- Map-reduce has high overhead, even a very simple operation on a small dataset can take in the hundreds of milliseconds (see example), so aggregation pipeline likely to be much faster for "small" data sets, Map-reduce likely to be faster for very large data sets as operations can run largely in parallel.
- Map-reduce uses JavaScript functions to define operations, so have the full power of the language available. Potentially can perform operations that the aggregation pipeline doesn't have operators for

Indexes

- An index is a data structure that stores information about the values of specified fields in the documents of a collection
- Indexes support the efficient execution of queries in MongoDB
- MongoDB supports three types of index:
  - Single field index
  - Compound index - can support queries that match on multiple fields
  - Multikey index – to index elements in an array field
- MongoDB creates a default index on the _id field during the creation of a collection

Creating indexes

```
{
    "_id": ObjectId(...),
    "item": "Banana",
    "category": ["food", "produce", "grocery"],
    "location": "4th Street Store",
    "stock": 4,
    "type": "cases",
    "ratings": [ 5, 8, 9 ]
}
```

- Create a single-field index on field *item*, descending order

```
db.products.createIndex({"item": -1})
```

- Create a compound index on fields *item* and *stock*, ascending

```
db.products.createIndex({ "item": 1, "stock": 1 } )
```

- Create a multikey index on field *ratings*, ascending

```
db.products.createIndex( { "ratings": 1 } )
```

Multikey indexes on embedded objects

```
{
  _id: 1,
  item: "abc",
  stock: [
    { size: "S", color: "red", quantity: 25 },
    { size: "S", color: "blue", quantity: 10 },
    { size: "M", color: "blue", quantity: 50 }
  ]
}
```

- Compound multikey index on *size* and *quantity* fields of array of embedded objects, where array is *stock* field of the parent object

```
db.inventory.createIndex( { "stock.size": 1, "stock.quantity": 1 } )
```

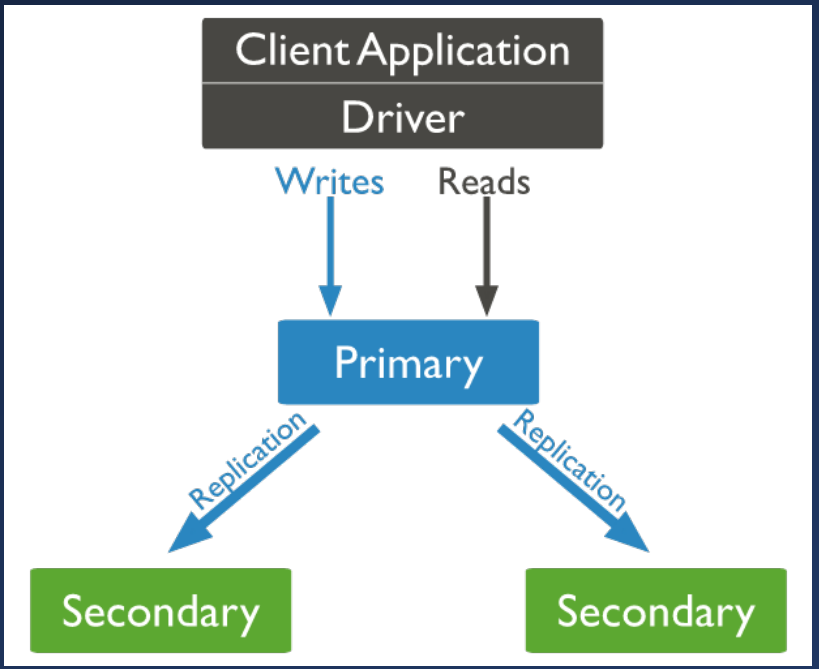
- MongoDB can be installed as a standalone database on a single computer. However, production installations typically involve clusters of computers.

- Core techniques used within a cluster:
  - Replication - provides redundancy and increase data availability
  - Sharding - to distribute a large set of data across multiple nodes, to support high throughput operations

- MongoDB can have clusters with 1000+ nodes

Replication

- A replica set in MongoDB is a group of mongod processes that maintain the same data set.
- Members of a replica set:
  - *Primary node* - receives all write operations
  - *Secondaries* - replicate operations from the primary to maintain an identical data set
  - If the primary is unavailable, an eligible secondary will hold an election to elect itself the new primary.
  - *Arbiter* - does not keep a copy of the data, but play a role in elections



- The minimum recommended configuration for a replica set is a three member replica set with three data-bearing members: one primary and two secondaries.

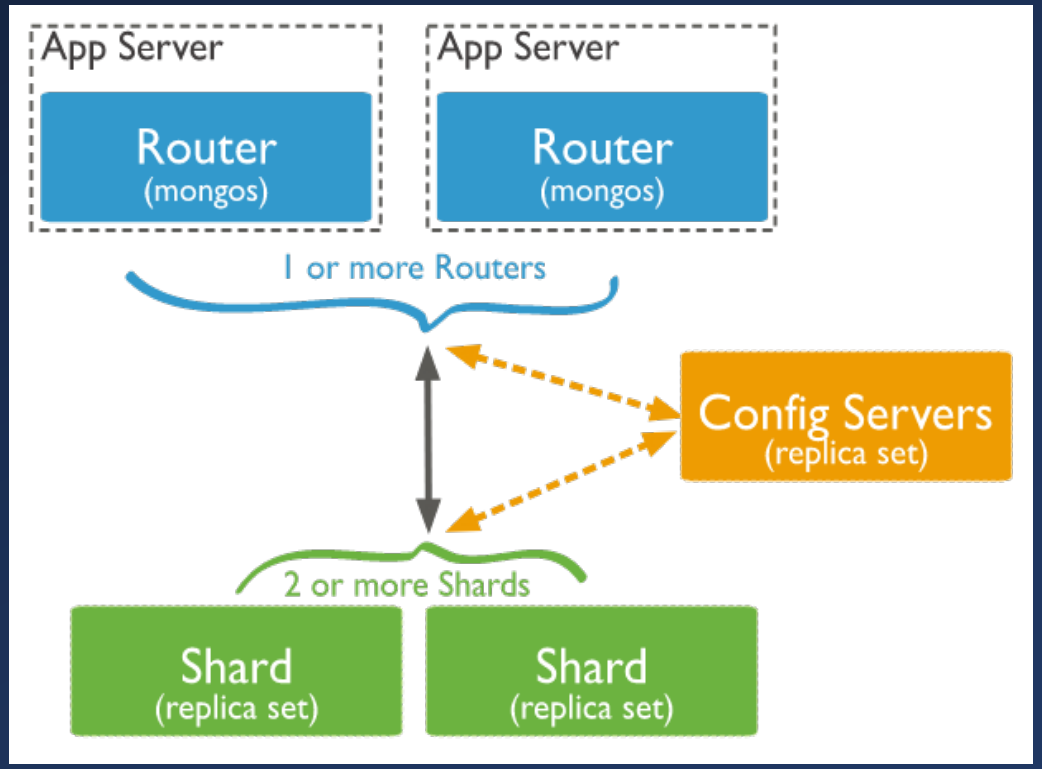
Sharding

- Sharding is a method for distributing data across multiple machines

A sharded cluster consists of:

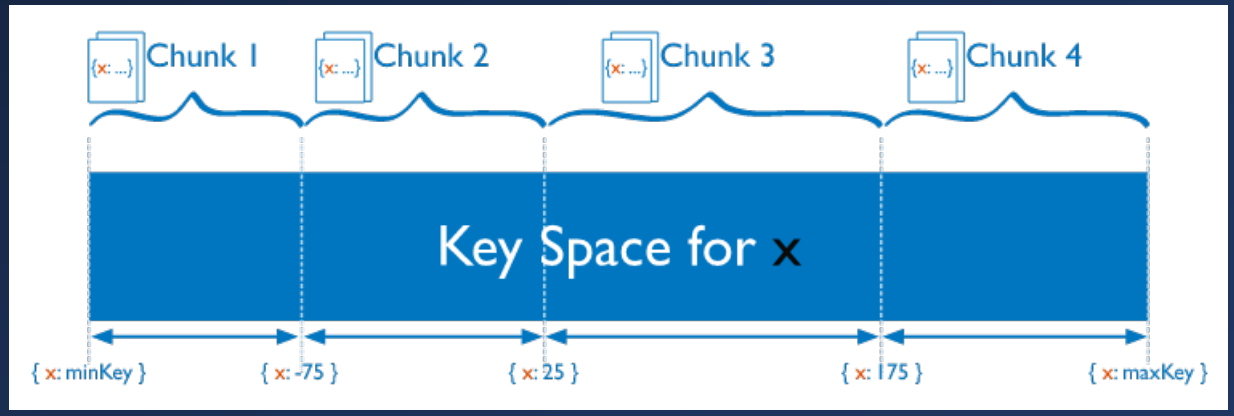
- *Shards*
  - Each shard contains a subset of the sharded data.
  - Each shard can be a replica set
- *Mongos*
  - Acts as a query router, providing an interface between client applications and the sharded cluster.
- *Config servers*
  - ◦ Store metadata and configuration settings

- MongoDB shards data at the collection level, distributing the collection data across the shards in the cluster.



Shard keys

- MongoDB uses the shard key to distribute the collection's documents across shards.
- The shard key consists of a field or multiple fields in the documents. To shard a populated collection, the collection must have an index that starts with the shard key.
- MongoDB partitions sharded data into chunks. Each chunk has an inclusive lower and exclusive upper range based on the shard key. MongoDB attempts to distribute chunks evenly among the shards in the cluster

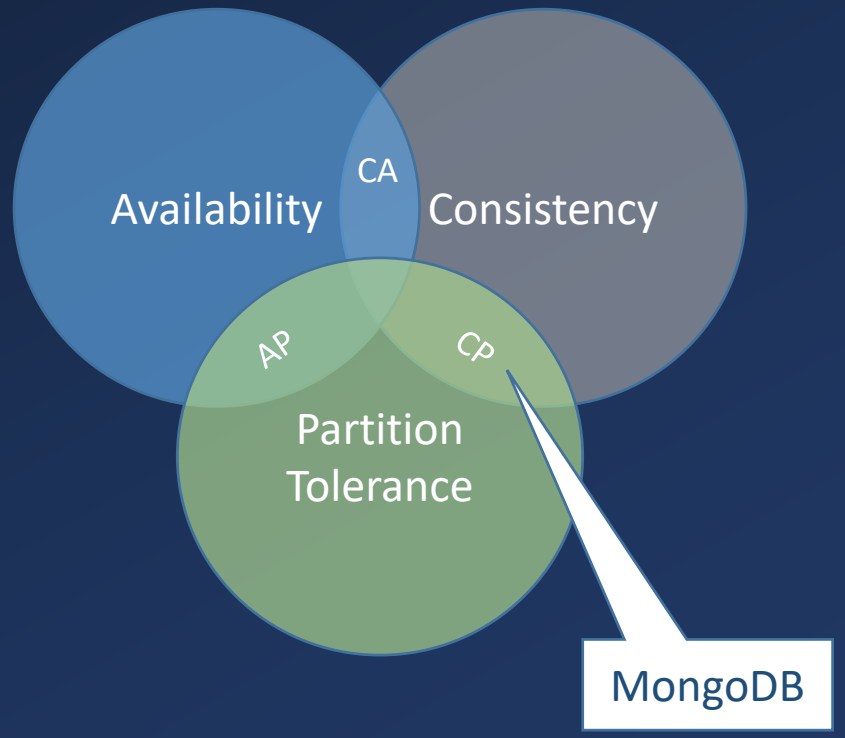


Shard keys

- The choice of shard key affects how the sharded cluster balancer creates and distributes chunks across the available shards.
- The ideal shard key allows MongoDB to distribute documents evenly throughout the cluster
- Shard key is immutable
- Considerations in choosing shard key:
  - *Cardinality* – the number of possible values, and hence the largest number of chunks
  - *Frequency* – how often a given value occurs in the data, key where a subset of values occur often can lead to uneven distribution and bottlenecks
  - *Rate of change* – if shard keys are monotonically increasing inserted data may be always routed to chunk with maxKey as upper bound, leading to bottlenecks

The CAP theorem with MongoDB

- By default, MongoDB is a strongly consistent system. Once a write completes, any subsequent read will return the most recent value.
- MongoDB gets high availability by replicating multiple copies of the data. The more copies, the higher the availability.
- With the CAP theorem MongoDB is categorized as a distributed system that guarantees consistency over availability



Summary

- Query on Embedded Documents
- Query Arrays
- Query Complex Objects
- Querying with Regular Expressions
- Aggregation Pipeline
- Map-Reduce
- Indexes
- Replication and sharding
- The CAP theorem with MongoDB