

CC5212-1

PROCESAMIENTO MASIVO DE DATOS

OTOÑO 2019

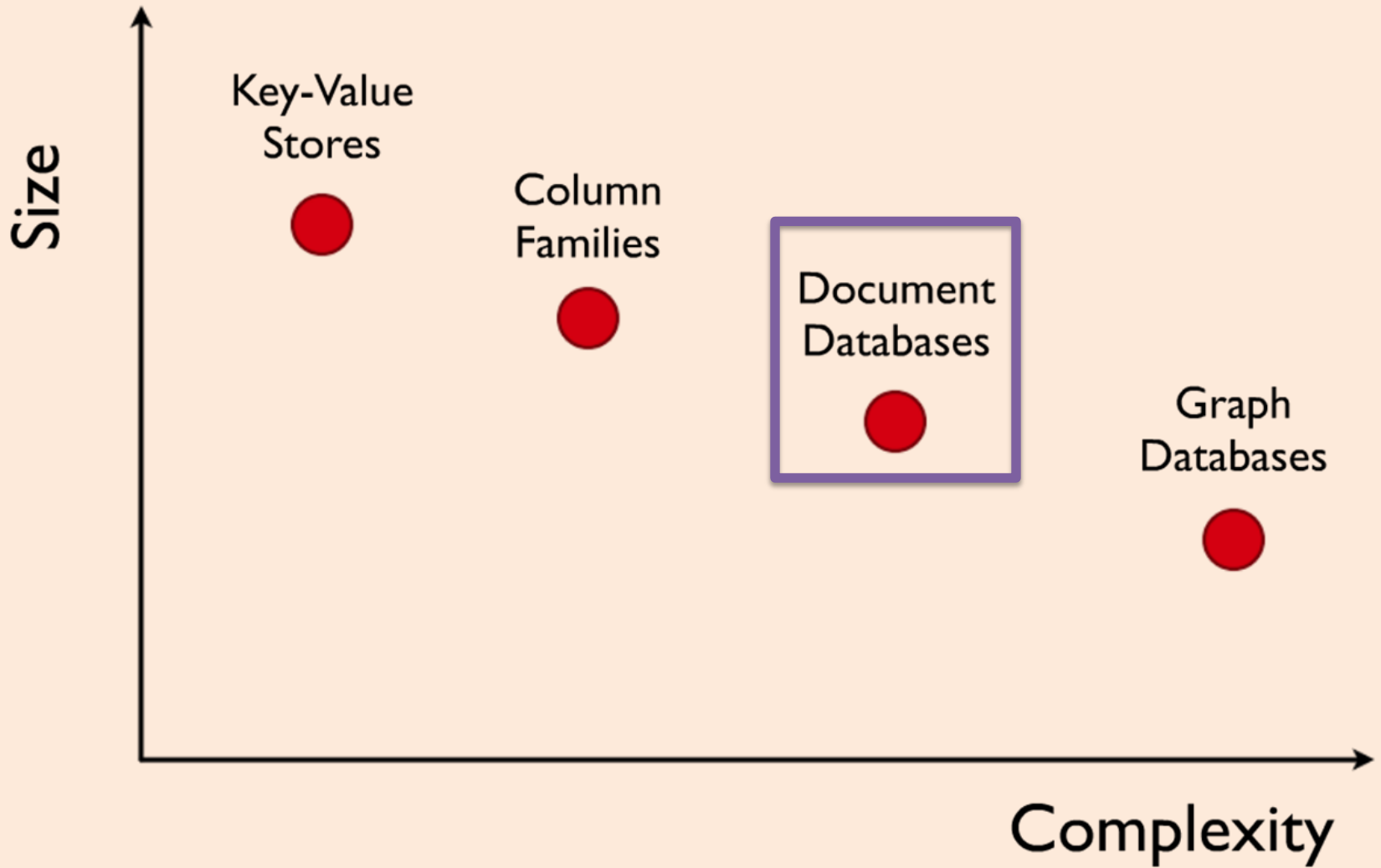
Lecture 10

NoSQL: MongoDB

Aidan Hogan

aidhog@gmail.com

NoSQL



DOCUMENT STORES

Key–Value: a Distributed Map

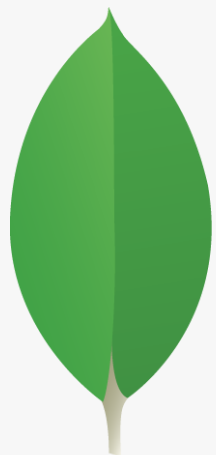
Countries	
Primary Key	Value
Afghanistan	capital:Kabul,continent:Asia,pop:31108077#2011
...	...

Tabular: Multi-dimensional Maps

Countries				
Primary Key	capital	continent	pop-value	pop-year
Afghanistan	Kabul	Asia	31108077	2011
...

Document: Value is a document

Countries	
Primary Key	Value
Afghanistan	{ cap: "Kabul", con: "Asia", pop: { val: 31108077, y: 2011 } }
...	...



mongoDB®

{JSON}

JavaScript Object Notation

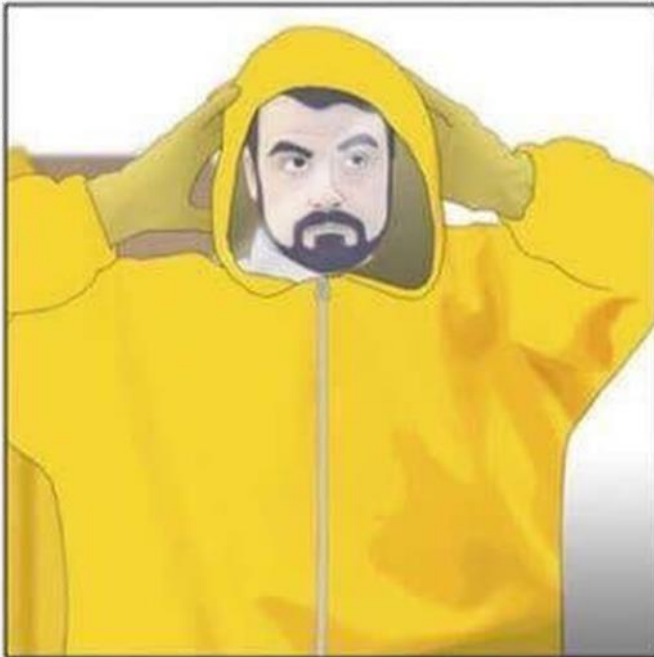
May 2019	Rank		DBMS	Database Model	Score		
	Apr 2019	May 2018			May 2019	Apr 2019	May 2018
1.	1.	1.	Oracle	Relational, Multi-model	1285.55	+5.61	-4.87
2.	2.	2.	MySQL	Relational, Multi-model	1218.96	+3.82	-4.38
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1072.19	+12.23	-13.66
4.	4.	4.	PostgreSQL	Relational, Multi-model	478.89	+0.17	+77.99
5.	5.	5.	MongoDB	Document	408.07	+6.10	+65.96
6.	6.	6.	IBM Db2	Relational, Multi-model	174.44	-1.61	-11.17
7.	8.	9.	Elasticsearch	Search engine, Multi-model	148.62	+2.62	+18.18
8.	7.	7.	Redis	Key-value, Multi-model	148.40	+2.03	+13.06
9.	9.	8.	Microsoft Access	Relational	143.78	-0.87	+10.67
10.	11.	10.	Cassandra	Wide column	125.72	+2.11	+7.89
11.	10.	11.	SQLite	Relational	122.90	-1.32	+7.44
12.	12.	14.	MariaDB	Relational, Multi-model	86.52	+1.29	+21.53
13.	13.	13.	Splunk	Search engine	85.24	+2.15	+20.15
14.	15.	18.	Hive	Relational	77.90	+3.19	+20.93
15.	14.	12.	Teradata	Relational	76.04	+0.69	+1.63
16.	16.	15.	Solr	Search engine	60.80	+0.57	-0.72
17.	17.	17.	HBase	Wide column	59.77	+1.11	-0.18
18.	18.	19.	FileMaker	Relational	58.51	+0.09	+3.84
19.	19.	21.	Amazon DynamoDB	Multi-model	55.93	-0.08	+11.74
20.	21.	20.	SAP HANA	Relational, Multi-model	55.74	+0.39	+7.37
21.	20.	16.	SAP Adaptive Server	Relational	55.44	-0.36	-6.07
22.	22.	22.	Neo4j	Graph	51.03	+1.54	+10.45
23.	23.	24.	Couchbase	Document	34.67	-1.61	+2.26
24.	25.	23.	Memcached	Key-value	28.90	+0.17	-4.66
25.	24.	26.	Microsoft Azure SQL Database	Relational, Multi-model	28.77	-0.02	+3.56

Why you should never, ever, ever use MongoDB

19 Jul 2015

MongoDB is evil. It...

- ... loses data (sources: [1](#), [2](#))
- ... in fact, for a long time, ignored errors by default and assumed every single write succeeded no matter what (which on 32-bits systems led to losing all data silently after some 3GB, due to MongoDB limitations)
- ... is slow, *even* at its advertised usecases, and claims to the contrary are completely lacking evidence (sources: [3](#), [4](#))
- ... forces the poor habit of implicit schemas in nearly all usecases (sources: [4](#))
- ... has locking issues (sources: [4](#))
- ... has an atrociously poor response time to security issues - it took them **two years** to patch an insecure default configuration that would expose *all of your data* to anybody who asked, without authentication (sources: [5](#))
- ... is not ACID-compliant (sources: [6](#))
- ... is a nightmare to scale and maintain
- ... isn't even exclusive in its offering of JSON-based storage; PostgreSQL does it too, and other (better) document stores like CouchDB have been around for a long time (sources: [7](#), [8](#))



MONGODB:

DATA MODEL

JavaScript Object Notation: JSON

```
{  
  "id": 179,  
  "name": "The Wire",  
  "type": "Scripted",  
  "language": "English",  
  "genres": [ "Drama", "Crime", "Thriller" ],  
  "status": "Ended",  
  "runtime": 60,  
  "premiered": "2002-06-02",  
  "schedule": {  
    "time": "21:00",  
    "days": [  
      "Sunday"  
    ]  
  },  
  "rating": {  
    "average": 9.4  
  }  
}
```



Binary JSON: BSON

```
{
  "_id": ObjectId(99a88b77c66d),
  "name": "The Wire",
  "type": "Scripted",
  "language": "English",
  "genres": [ "Drama", "Crime", "Thriller" ],
  "status": "Ended",
  "runtime": 60,
  "premiered": ISODate("2002-06-02"),
  "schedule": {
    "time": "21:00",
    "days": [
      "Sunday"
    ]
  },
  "rating": {
    "average": 9.4
  }
}
```

BSON { 01010100
11101011
10101110
01010101 }

MongoDB: Datatypes

String: "sí"

Boolean: true

Array: []

Double: 43.2

Object: {}

{JSON}
JavaScript Object Notation
etc.

ObjectID: ObjectId(0A0A0A0A0A0A)

Date: ISODate("2003-14-15T09:26:53.589Z")

BSON {
01010100
11101011
10101110
01010101
etc.

MongoDB: Map from keys to BSON values

TVSeries	
Key	BSON Value
99a88b77c66d	<pre>{ "_id": ObjectId(99a88b77c66d), "name": "The Wire", "type": "Scripted", "language": "English", "genres": ["Drama", "Crime", "Thriller"], "status": "Ended", "runtime": 60, "premiered": ISODate("2002-06-02"), "schedule": { "time": "21:00", "days": ["Sunday"] }, "rating": { "average": 9.4 } }</pre>
...	...

MongoDB Collection: Similar Documents

TVSeries	
Key	BSON Value
99a88b77c66d	<pre>{ "_id": ObjectId(99a88b77c66d), "name": "The Wire", "type": "Scripted", ... }</pre>
11f22e33d44c	<pre>{ "_id": ObjectId(11f22e33d44c), "name": "Rick and Morty", "type": "Animation", ... }</pre>

MongoDB Database: Related Collections

TVSeries

Key	BSON Value
...	...

TVEpisodes

Key	BSON Value
...	...

TVNetworks

Key	BSON Value
...	...

database: TV

Load database tvdb (and create if not exists):

```
> use tvdb
```

```
switched to db tvdb
```

See all (non-empty) databases (tvdb is empty):

```
> show dbs
```

```
local      0.00001 GB  
test      0.00231 GB
```

See current database:

```
> db
```

```
tvdb
```

Drop current database:

```
> db.dropDatabase()
```

```
{ "dropped" : "tvdb", "ok" : 1 }
```


Create collection series:

```
> db.createCollection("series")
```

```
{ "ok" : 1 }
```

See all collections in current database:

```
> show collections
```

```
series
```

Drop collection series:

```
> db.series.drop()
```

```
true
```

Clear all documents from collection series:

```
> db.series.remove( {} )
```

```
WriteResult({ "nRemoved" : 0 })
```

Create capped collection (keeps only most recent):

```
> db.createCollection("last100episodes",  
  { capped: true, size: 12285600, max: 100 } )  
  
{ "ok" : 1 }
```

Create collection with default index on `_id`:

```
> db.createCollection("cast", { autoIndexId: true } )  
  
{  
  "note" : "the autoIndexId option is deprecated ...",  
  "ok" : 1  
}
```

MONGODB: INSERTING DATA

Insert document: without `_id`

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60,  
    "genres": [  
      "Science-Fiction",  
      "Thriller"  
    ]  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

Insert document: with `_id`

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60,  
    "genres": [  
      "Science-Fiction",  
      "Thriller"  
    ]  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

... fails if `_id` already exists

... use `update` or `save` to update existing document(s)

Use save and `_id` to overwrite:

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.save(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror (Overwritten)"  
  })
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

... overwrites old document

MONGODB: QUERIES WITH SELECTION

Query documents in a collection with `find`:

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

```
> db.series.find()  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "type" : "Scripted", "runtime" : 60 }
```

... use `findOne()` to return one document

Pretty print results with pretty:

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

```
> db.series.find().pretty()  
  
{  
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),  
  "name" : "Black Mirror",  
  "type" : "Scripted",  
  "runtime" : 60  
}
```

Selection: find documents matching σ

```
> db.series.find( $\sigma$ )
```

Selection σ : Equality

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "type": "Scripted",  
    "runtime": 60  
  })
```

Equality: { key: value }

```
> db.series.find( { "type": "Scripted" } )  
{ "name" : "Black Mirror", "type" : "Scripted", "runtime" : 60 }
```

Results would include `_id` but for brevity, we will omit this from examples where it is not important.



Selection σ : Nested key

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4 },  
    "runtime": 60  
  })
```

Key can access nested values:

```
> db.series.find( { "rating.avg": 9.4 } )  
  
{ "name" : "Black Mirror", "rating": { "avg": 9.4 }, "runtime" : 60 }
```

Selection σ : Equality on null

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": null },  
    "runtime": 60  
  })
```

Equality on nested null value:

```
> db.series.find( { "rating.avg": null } )  
  
{ "name" : "Black Mirror", "rating": { "avg": null }, "runtime" : 60 }
```

... matches when value is null or ...

Selection σ : Equality on null

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "val": 9.4 },  
    "runtime": 60  
  })
```

Key can access nested values:

```
> db.series.find( { "rating.avg": null } )  
  
{ "name" : "Black Mirror", "rating": { "val": 9.4 }, "runtime" : 60 }
```

... when field doesn't exist with key.

Selection σ : Equality on document

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4 },  
    "runtime": 60  
  })
```

Value can be an object/document:

```
> db.series.find( { "rating": { "avg": 9.4 } } )  
{ "name" : "Black Mirror", "rating": { "avg": 9.4 }, "runtime" : 60 }
```

Selection σ : Equality on document

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4, "votes": 9001 },  
    "runtime": 60  
  })
```

Value can be an object/document:

```
> db.series.find( { "rating": { "votes":9001 } } )
```

... no results: needs to match full object.

Selection σ : Equality on document

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4, "votes": 9001 },  
    "runtime": 60  
  })
```

Value can be an object/document:

```
> db.series.find( { "rating": { "votes":9001, "avg":9.4 } } )
```

... no results: order of attributes matters.

Selection σ : Equality on exact array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality can match an exact array:

```
> db.series.find( { "genres": [ "Science-Fiction",  
                               "Thriller" ] } )  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Equality on exact array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality can match an exact array

```
> db.series.find( { "genres": [ "Science-Fiction" ] } )
```

... no results: needs to match full array.

Selection σ : Equality on exact array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality can match an exact array

```
> db.series.find( { "genres": [ "Thriller",  
                               "Science-Fiction" ] } )
```

... no results: order of elements matters.

Selection σ : Equality matches inside array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Equality matches a value in an array:

```
> db.series.find( { "genres": "Thriller" } )  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Equality matches both

```
> db.series.insert( { "name": "A" , "val": [ 5, 6 ] } )  
> db.series.insert( { "name": "B" , "val": 5 } )
```

Equality matches a value inside and outside an array:

```
> db.series.find( { "val": 5 } )  
  
{ "name": "A" , "val": [ 5, 6 ] }  
{ "name": "B" , "val": 5 }
```

cough

Selection σ : Inequalities

Less than: `{ key: { $lt: value } }`

Greater than: `{ key: { $gt: value } }`

Less than or equal: `{ key: { $lte: value } }`

Greater than or equal: `{ key: { $gte: value } }`

Not equals: `{ key: { $ne: value } }`

Selection σ : Less Than

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Less than: { key: { \$lt: value } }

```
> db.series.find({ "runtime": { $lt: 70 } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```


Selection σ : Match one of multiple values

Match any value: { key: { \$in: [v1, ..., vn] } }

Match no value: { key: { \$nin: [v1, ..., vn] } }

... also passes if key does not exist.

Selection σ : Match one of multiple values

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Match any value: { key: { \$in: [v1, ..., vn] } }

```
> db.series.find({ "runtime": { $in: [30, 60] } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Match one of multiple values

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Match any value: { key: { \$in: [v1, ..., vn] } }

```
> db.series.find({ "genres": { $in: ["Noir", "Thriller"] } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

... if key references an array, any value of array should match any value of \$in

Selection σ : Boolean connectives

And: $\{ \$and: [\sigma , \sigma'] \}$

Or: $\{ \$or: [\sigma , \sigma'] \}$

Not: $\{ \$not: [\sigma] \}$

Nor: $\{ \$nor: [\sigma , \sigma'] \}$

... of course, can nest such conditions

Selection σ : And

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

And: $\{ \$and: [\sigma , \sigma'] \}$

```
> db.series.find({ $and: [  
  { "runtime": { $in: [30, 60] } } ,  
  { "name": { $ne: "Lost" } } ] }  
)  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Attribute (not) exists

Exists: { key: { \$exists : true } }

Not Exists: { key: { \$exists : false } }

Selection σ : Attribute exists

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Exists: { key: { \$exists : true } }

```
> db.series.find({ "name": { $exists : true } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

... checks that the field key exists
(even if value is NULL)

Selection σ : Attribute not exists

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Not exists: { key: { \$exists : false } }

```
> db.series.find({ "name": { $exists : false } })
```

... checks that the field key doesn't exist
(empty results)

Selection σ : Arrays

All: `{ key: { $all : [v1, ..., vn] } }`

Match one: `{ key: { $elemMatch : { σ 1, ..., σ n } } }`

Size: `{ key: { $size : int } }`

Selection σ : Array contains (at least) all elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Sci-Fi", "Thriller", "Comedy" ],  
    "runtime": 60  
  })
```

All: `{ key: { $all : [v1, ..., vn] } }`

```
> db.series.find(  
  { "genres": { $all : [ "Comedy", "Sci-Fi" ] } })  
  
{ "name" : "Black Mirror", "genres": [ "Sci-Fi", "Thriller", "Comedy" ],  
  "runtime" : 60 }
```

... all values are in the array

Selection σ : Array element matches (with AND)

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Match one: { key: { \$elemMatch : { σ_1 , ..., σ_n } } }

```
> db.series.find(  
  { "series": { $elemMatch : { $gt: 1, $lt: 3 } } })  
  
{ "name" : "Black Mirror", "series": [ 1, 2, 3 ], "runtime" : 60 }
```

... one element matches all criteria (with AND)

Selection σ : Array with exact size

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Size: `{ key: { $size : int } }`

```
> db.series.find( { "series": { $size : 3 } })  
{ "name" : "Black Mirror", "series": [ 1, 2, 3 ], "runtime" : 60 }
```

... only possible for exact size of array (not ranges)

Selection σ : Type of value

Type: `{ key: { $type: typename } }`

`"timestamp"`
`"double"`
`"date"`
`"array"`
`"objectId"`
`"dbPointer"`
`"javascript"`
`"string"`
`"object"`
`"bool"`
`"null"`
`"number"`
`"decimal"`
`"binData"`
`"int"`
`"long"`
`"undefined"`
`"regex"`
`"array"`
`"number"`
`...`

Selection σ : Type of value

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Type: { key: { \$type: typename } }

```
> db.series.find({ "runtime": { $type : "number" } })  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller" ],  
  "runtime" : 60 }
```

Selection σ : Matching an array by type?

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

Type: { key: { \$type: typename } }

```
> db.series.find({ "genres": { $type : "array" } })
```

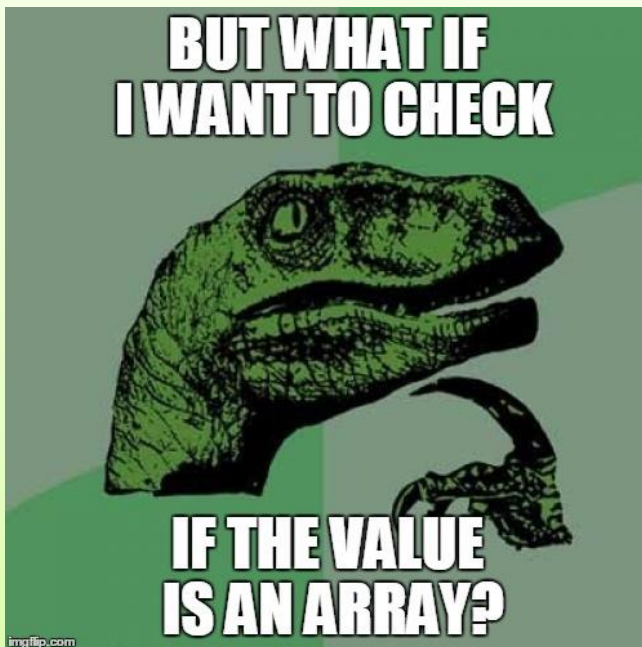
... empty
... passes if any value in the array has that type

Selection σ : Matching an array by type?

Arrays

When applied to arrays, `$type` matches any **inner** element that is of the specified **BSON** type. For example, when matching for `$type : 'array'`, the document will match if the field has a nested array. It will not return results where the field itself is an array.

<https://docs.mongodb.com/manual/reference/operator/query/type/>



```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [ "Science-Fiction", "Thriller" ],  
    "runtime": 60  
  })
```

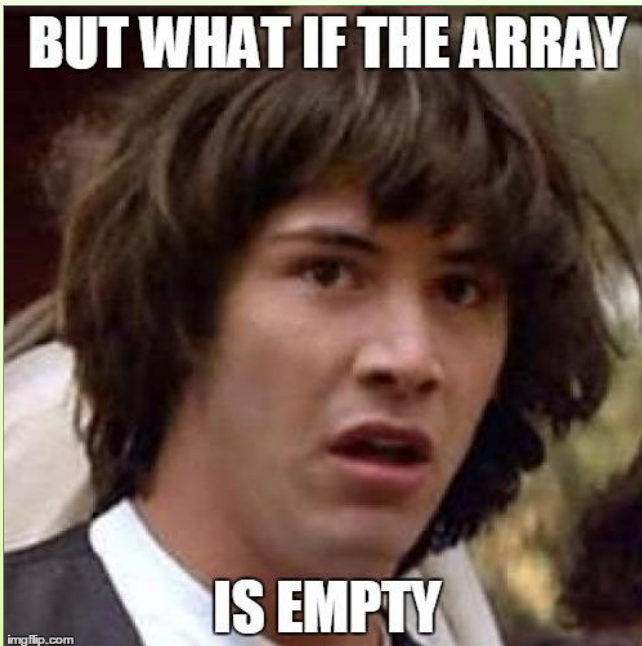
```
> db.series.find(  
  { "genres": { $elemMatch: { $exists : true } } }  
)  
  
{ "name" : "Black Mirror", "genres": [ "Science-Fiction", "Thriller"  
], "runtime" : 60 }
```


Selection σ : Matching an array by type?

Arrays

When applied to arrays, `$type` matches any **inner** element that is of the specified **BSON** type. For example, when matching for `$type : 'array'`, the document will match if the field has a nested array. It will not return results where the field itself is an array.

<https://docs.mongodb.com/manual/reference/operator/query/type/>



```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "genres": [],  
    "runtime": 60  
  })
```

```
> db.series.find(  
  { $or: [  
    { "genres": { $elemMatch: { $exists: true } } },  
    { "genres": [] }  
  ] } )
```

```
{ "name" : "Black Mirror", "genres": [], "runtime" : 60 }
```



SEEMS LEGIT

Selection σ : Other operators

Mod: `{ key: { $mod [div, rem] } }`

Regex: `{ key: { $regex: pattern } }`

Text search: `{ $text: { $search: terms } }`

Where (JS): `{ $where: javascript_code }`

... where is executed over all documents
(and should be avoided where possible)

Selection σ : Geographic features



Selection σ : Bitwise features

`$bitsAllClear`

`$bitsAllSet`

`$bitsAnyClear`

`$bitsAnySet`

...

<https://docs.mongodb.com/manual/reference/operator/query-bitwise/>

MONGODB:

PROJECTION OF OUTPUT VALUES

Projection π : Choose output values

<code>key: 1:</code>	Output field(s) with <code>key</code>
<code>key: 0:</code>	Suppress field(s) with <code>key</code>
<code>array.\$: 1</code>	Project first matching array element I
<code>\$elemMatch:</code>	Project first matching array element II
<code>\$slice:</code>	Output first or last <code>slice</code> array values

Projection π : Output certain fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project only certain fields: $\{ k1: 1, \dots, kn: 1 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "name": 1 , "runtime": 1, "network": 1 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "runtime" : 60 }
```

... outputs what is available; by default also outputs `_id` field

Projection π : Output embedded fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "rating": { "avg": 9.4 , "votes": 9001 },  
    "runtime": 60  
  })
```

Project (embedded) fields: $\{ k1: 1, \dots, kn: 1 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "name": 1 , "rating.avg": 1 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "rating" : { "avg": 9.4 } }
```

... field is still nested in output

Projection π : Output embedded fields in array

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "reviews": [  
      { "user": "jack" , "score": 9.1 },  
      { "user": "jill" , "score": 8.3 }  
    ],  
    "runtime": 60  
  })
```

Project (embedded) fields: $\{ k1: 1, \dots, kn: 1 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "name": 1 , "reviews.score": 1 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "reviews" : [ { "score": 9.1 } , { "score": 8.3 } ] }
```

... projects from within the array.

Projection π : Suppress certain fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Return all but certain fields: $\{ k_1: 0, \dots, k_n: 0 \}$

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "series": 0 })  
  
{ "_id" : ObjectId("5951e0b265ad257d48f4a7d5"), "name" : "Black Mirror",  
  "runtime" : 60 }
```

... cannot combine 0 and 1 except ...

Projection π : Suppress certain fields

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Suppress ID: `{ _id: 0, k1: 1, ..., kn: 1 }`

```
> db.series.find(  
  { "runtime": { $gt: 30 } },  
  { "_id": 0, "name": 1, "series": 1 })  
  
{ "name" : "Black Mirror", "series" : [ 1, 2, 3 ] }
```

... 0 suppresses `_id` when other fields are output

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project first matching element: `array.$: 1`

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series.$": 1 } )  
  
{ _id: ..., "series": [ 2 ] }
```

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project first matching element: $\$elemMatch$

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $elemMatch: { $lt: 3 } } } )  
  
{ "_id": ..., "series": [ 1 ] }
```

... allows to separate selection and projection criteria.

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Project first matching element: $\$elemMatch$

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $elemMatch: { $gt: 3 } } } )  
  
{ "_id": ... }
```

... drops array field entirely if no element is projected.

Projection π : Output first matching element

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "reviews": [  
      { "user": "jack" , "score": 9.1 },  
      { "user": "jill" , "score": 8.3 }  
    ]  
  }  
)
```

Project first matching element: `$elemMatch`

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "reviews": { $elemMatch: { "score": { $gt: 8 } } } } )  
  
{ "_id": ..., "reviews": [ { "user": "jack" , "score": 9.1 } ] }
```

... can match on array of documents.

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Return first n elements: $\$slice: n$ (where $n > 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: 2 } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 1, 2 ], "runtime": 60 }
```

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Return last n elements: $\$slice: n$ (where $n < 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: -2 } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 2, 3 ], "runtime": 60 }
```

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

Skip n and return m: $\$slice: [n,m]$ ($n, m > 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: [ 2, 1 ] } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 3 ], "runtime": 60 }
```

Projection π : Output some matching elements

```
> db.series.insert(  
  {  
    "name": "Black Mirror",  
    "series": [ 1, 2, 3 ],  
    "runtime": 60  
  })
```

From last n , return m : `$slice: [n,m]` ($n < 0, m > 0$)

```
> db.series.find(  
  { "series": { $elemMatch: { $gt: 1 } } },  
  { "series": { $slice: [ -2, 1 ] } } )  
  
{ "_id": ..., "name": "Black Mirror", "series": [ 2 ], "runtime": 60 }
```

MONGODB:

UPDATES

Update **u**: Modify fields in documents

\$set :	Set the value
\$unset :	Remove the key and value
\$rename :	Rename the field (change the key)
\$setOnInsert :	You don't want to know
\$inc :	Increment number by inc
\$mul :	Multiply number by mul
\$min :	Replace values less than min by min
\$max :	Replace values greater than max by max
\$currentDate :	Set to current date

Use update with **query** criteria and **update** criteria:

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "type": "Scripted",
    "language": "English",
  })

WriteResult({ "nInserted" : 1 })

> db.series.update(
  { "type": "Scripted" },
  { $set: { "type": "Fiction" } },
  { "multi": true }
)

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.series.find({ "name": "Black Mirror" })

{
  "_id": ObjectId("5951e0b265ad257d48f4a7d5"),
  "name": "Black Mirror",
  "type": "Fiction",
  "language": "English"
}
```

Update **u**: Modify arrays in documents

- \$addToSet**: Adds value if not already present
- \$pop**: Deletes first or last value
- \$push**: Appends (an) item(s) to the array
- \$pullAll**: Removes values from a list
- \$pull**: Removes values that match a condition

(Sub-)operators used for pushing/adding values:

- \$**: Select first element matching query condition
- \$each**: Add or push multiple values
- \$slice**: After pushing, keep first or last **slice** values
- \$sort**: Sort the array after pushing
- \$position**: Push values to a specific array index

Update **u**: Modify arrays in documents

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "type": [ "Scripted", "Drama" ],  
    "languages": [ "English", "Spanish" ],  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.update(  
  { "type": "Scripted" },  
  { $pull: { "type": { $in: [ "Drama", "Sci-Fi" ] } },  
    "languages": "Spanish" } },  
  { "multi": true } )
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.series.find({ "name": "Black Mirror" })  
  
{  
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),  
  "name" : "Black Mirror",  
  "type" : [ "Scripted" ],  
  "languages" : [ "English" ] }
```

Update **u**: Modify arrays in documents

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "ratings": [ 8, 11, 13, 9 ],  
    "languages": [ "English", "Spanish" ],  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.update(  
  { "ratings": { $gt: 10 } },  
  { $set: { "ratings.$": 10 } },  
  { "multi": true } )
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

```
> db.series.find({ "name": "Black Mirror" })  
  
{  
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),  
  "name" : "Black Mirror",  
  "ratings" : [ 8, 10, 13, 9 ],  
  "languages" : [ "English", "Spanish" ] }
```

Update **u**: Modify arrays in documents

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "ratings": [ 8, 11, 13, 9 ],
    "languages": [ "English", "Spanish" ],
  })

WriteResult({ "nInserted" : 1 })

> db.series.update(
  { "ratings": { $gt: 10 } },
  { $push: { "ratings": { $each: [4, 9],
                                $sort: 1, $slice: 4 } } },
  { "multi": true } )

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.series.find({ "name": "Black Mirror" })

{
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),
  "name" : "Black Mirror",
  "ratings" : [ 4, 8, 9, 9 ],
  "languages" : [ "English", "Spanish" ] }
```

MONGODB:

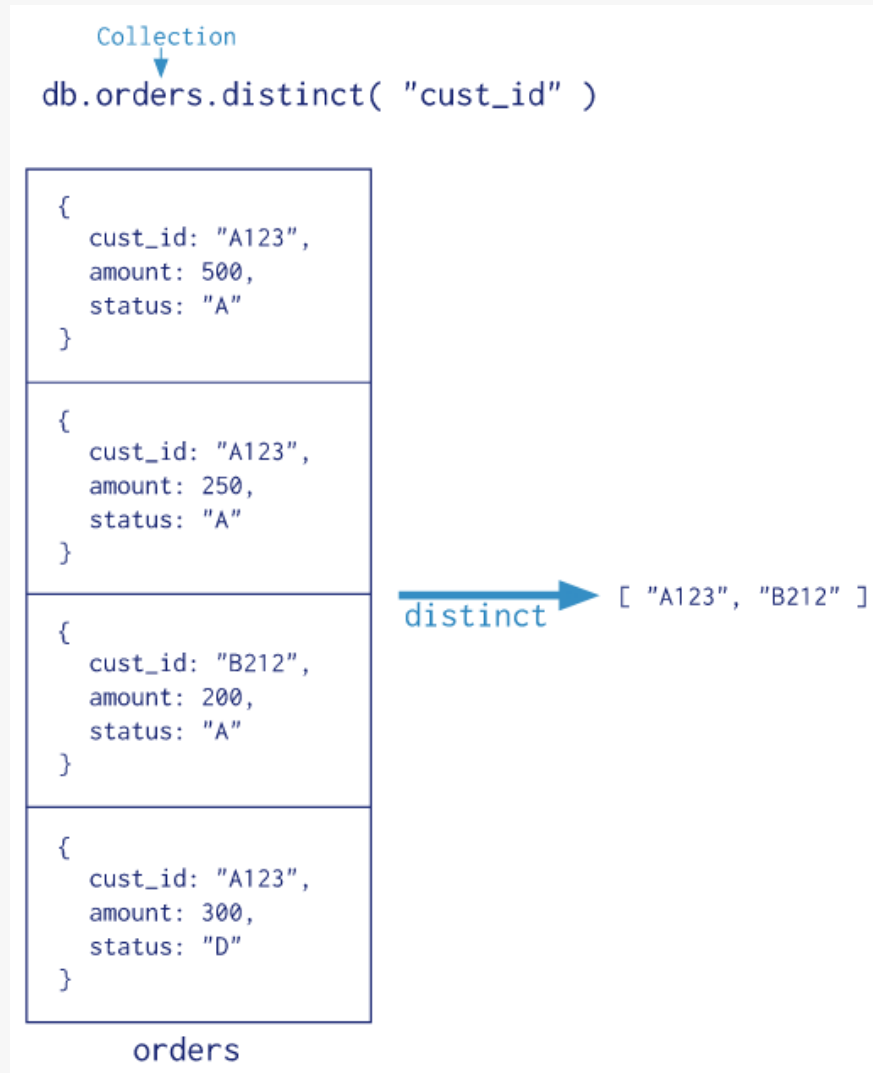
AGGREGATION AND PIPELINES

Aggregation: without grouping

`db.coll.distinct(key)`: Array of unique values for that key
<https://docs.mongodb.com/manual/reference/method/db.collection.distinct/>

`db.coll.count()`: Count the documents
<https://docs.mongodb.com/manual/reference/method/db.collection.count/>

Aggregation: distinct



Pipelines: Transforming Collections



Pipeline ρ : Stage Operators

\$match:	Filter by selection criteria σ
\$project:	Perform a projection π
\$group:	Group documents by a key/value [used with <i>\$sum</i> , <i>\$avg</i> , <i>\$first</i> , <i>\$last</i> , <i>\$max</i> , <i>\$min</i> , etc.]
\$lookup:	Perform left-outer-join with another collection
\$unwind:	Copy each document for each array value
\$collStats:	Get statistics about collection
\$count:	Count the documents in the collection
\$sort:	Sort documents by a given key (ASC DESC)
\$limit:	Return (up to) n first documents
\$sample:	Return (up to) n sampled documents
\$skip:	Skip n documents
\$out:	Save collection to MongoDB

(more besides)

<https://docs.mongodb.com/manual/reference/operator/aggregation/>

Pipeline Aggregation: `$match` and `$group`

Collection
↓
`db.orders.aggregate([`
 `$match stage` → `{ $match: { status: "A" } },`
 `$group stage` → `{ $group: { _id: "$cust_id", total: { $sum: "$amount" } } }`
 `]`)

{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }

orders

→ `$match`

{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }

→ `$group`

Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

Pipeline Aggregation: \$lookup

```
{
  "name": "The Wire",
  "country": "US"
}
{
  "name": "Black Mirror",
  "country": "UK"
}
```

```
{
  "nation": "US",
  "network": "HBO"
}
{
  "nation": "US",
  "network": "FOX"
}
```

```
> db.series.aggregate( [
  { $lookup: { from: "networks", localField: "country",
    foreignField: "nation", as: "possibleNetworks" }
} ] )
```

```
{
  "name": "The Wire",
  "country": "US",
  "possibleNetworks": [
    { "nation": "US", "network": "HBO" },
    { "nation": "US", "network": "FOX" }
  ]
}
{
  "name": "Black Mirror",
  "country": "UK",
  "possibleNetworks": []
}
```

Pipeline Aggregation: \$unwind

```
{  
  "name": "The Wire",  
  "genres": [ "Drama", "Crime", "Thriller" ]  
}
```

```
> db.series.aggregate( [ { $unwind : "$genres" } ] )
```

```
{  
  "name": "The Wire",  
  "genres": "Drama"  
}  
{  
  "name": "The Wire",  
  "genres": "Crime"  
}  
{  
  "name": "The Wire",  
  "genres": "Thriller"  
}
```

MONGODB:

INDEXING

MongoDB Indexing

Per collection:

- `_id` Index
- Single-field Index (sorted)
- Compound Index (sorted)
- Multikey Index (for arrays)
- Geospatial Index
- Full-text Index
- Hash-based indexing (hashed)

MongoDB: Text Indexing Example

```
> db.series.insert(  
  {  
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),  
    "name": "Black Mirror",  
    "summary": "A British sci-fi series with a dystopian setting",  
    "languages": [ "English", "Spanish" ],  
  })
```

```
WriteResult({ "nInserted" : 1 })
```

```
> db.series.createIndex( { summary: "text"})
```

```
> db.series.getIndexes()
```

```
[  
  {  
    "v" : 2,  
    "key" : { "_id" : 1 },  
    "name" : "_id_",  
    "ns" : "test.series"      },  
  {  
    "v" : 2,  
    "key" : { "fts" : "text", "_ftsx" : 1 },  
    "name" : "summary_text",  
    "ns" : "test.series",  
    "weights" : { "summary" : 1 },  
    "default_language" : "english",  
    "language_override" : "language",  
    "textIndexVersion" : 3      }  
]
```

MongoDB: Text Indexing Example

```
> db.series.insert(
  {
    _id: ObjectId("5951e0b265ad257d48f4a7d5"),
    "name": "Black Mirror",
    "summary": "A British sci-fi series with a dystopian setting",
    "languages": [ "English", "Spanish" ],
  })

WriteResult({ "nInserted" : 1 })

> db.series.createIndex( { summary: "text" })

> db.series.find(
  { $text: { $search: "dystopia sci-fi" } } )

{
  "_id" : ObjectId("5951e0b265ad257d48f4a7d5"),
  "name" : "Black Mirror",
  "summary" : "A British sci-fi series with a dystopian setting",
  "languages" : [
    "English",
    "Spanish"
  ]
}
```

MONGODB:

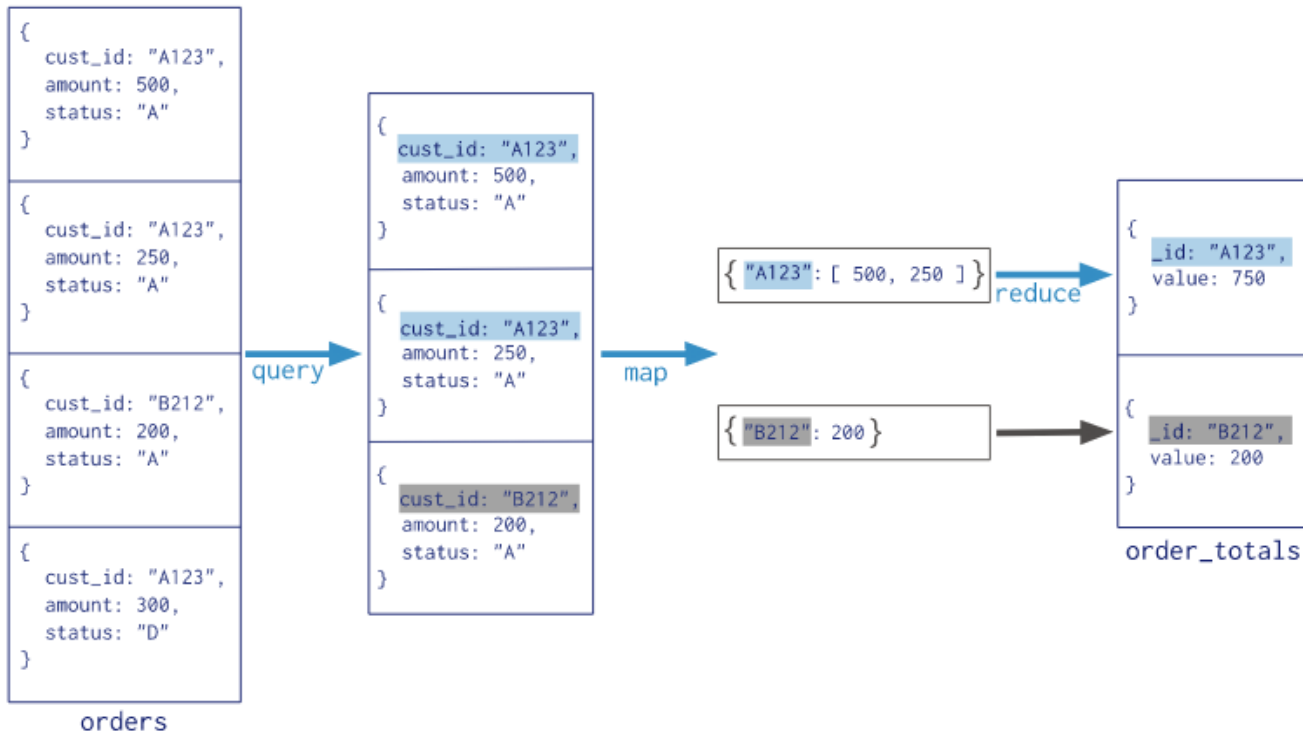
DISTRIBUTION

MongoDB: Distribution

- "Sharding":
 - Hash-based or Horizontal Ranged
(Depends on indexes)
- Replication
 - Replica sets

mapreduce

```
Collection
↓
db.orders.mapReduce(
  map   → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ) },
  query → {
    query: { status: "A" },
    out: "order_totals"
  }
)
```



MONGODB:

WHY IS IT SO POPULAR?

May 2019	Rank		DBMS	Database Model	Score		
	Apr 2019	May 2018			May 2019	Apr 2019	May 2018
1.	1.	1.	Oracle	Relational, Multi-model	1285.55	+5.61	-4.87
2.	2.	2.	MySQL	Relational, Multi-model	1218.96	+3.82	-4.38
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1072.19	+12.23	-13.66
4.	4.	4.	PostgreSQL	Relational, Multi-model	478.89	+0.17	+77.99
5.	5.	5.	MongoDB	Document	408.07	+6.10	+65.96
6.	6.	6.	IBM Db2	Relational, Multi-model	174.44	-1.61	-11.17
7.	8.	9.	Elasticsearch	Search engine, Multi-model	148.62	+2.62	+18.18
8.	7.	7.	Redis	Key-value, Multi-model	148.40	+2.03	+13.06
9.	9.	8.	Microsoft Access	Relational	143.78	-0.87	+10.67
10.	11.	10.	Cassandra	Wide column	125.72	+2.11	+7.89
11.	10.	11.	SQLite	Relational	122.90	-1.32	+7.44
12.	12.	14.	MariaDB	Relational, Multi-model	86.52	+1.29	+21.53
13.	13.	13.	Splunk	Search engine	85.24	+2.15	+20.15
14.	15.	18.	Hive	Relational	77.90	+3.19	+20.93
15.	14.	12.	Teradata	Relational	76.04	+0.69	+1.63
16.	16.	15.	Solr	Search engine	60.80	+0.57	-0.72
17.	17.	17.	HBase	Wide column	59.77	+1.11	-0.18
18.	18.	19.	FileMaker	Relational	58.51	+0.09	+3.84
19.	19.	21.	Amazon DynamoDB	Multi-model	55.93	-0.08	+11.74
20.	21.	20.	SAP HANA	Relational, Multi-model	55.74	+0.39	+7.37
21.	20.	16.	SAP Adaptive Server	Relational	55.44	-0.36	-6.07
22.	22.	22.	Neo4j	Graph	51.03	+1.54	+10.45
23.	23.	24.	Couchbase	Document	34.67	-1.61	+2.26
24.	25.	23.	Memcached	Key-value	28.90	+0.17	-4.66
25.	24.	26.	Microsoft Azure SQL Database	Relational, Multi-model	28.77	-0.02	+3.56



Questions?